

Lecture 14: Joins!

Announcements: Two Hints

- You may want to do *Trigger activity* for project 2.
 - We've noticed those who do it have less trouble with project!
 - Seems like we're good here 😊 Exciting for us!
- We posted an activity for you to do on your own... it may overlap heavily with a ps #3 problem... (*this is not necessary but helpful*).
 - The solutions will **not** be posted.
- Sorry the Google lecture was not recorded! Last minute thing...

1. Nested Loop Joins

What you will learn about in this section

1. RECAP: Joins
2. Nested Loop Join (NLJ)
3. Block Nested Loop Join (BNLJ)
4. Index Nested Loop Join (INLJ)

RECAP: Joins

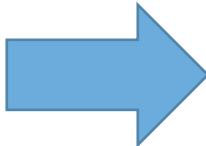
Joins: Example

 $R \bowtie S$

```
SELECT R.A, B, C, D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples $r \in R, s \in S$ such that $r.A = s.A$

R			S					
A	B	C	A	D				
1	0	1	3	7				
2	3	4	2	2				
2	5	2	2	3				
3	1	1						



A	B	C	D
2	3	4	2

Joins: Example

 $R \bowtie S$

```
SELECT R.A, B, C, D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples $r \in R, s \in S$ such that $r.A = s.A$

R			S					
A	B	C	A	D				
1	0	1	3	7				
2	3	4	2	2				
2	5	2	2	3				
3	1	1			A	B	C	D
					2	3	4	2
					2	3	4	3

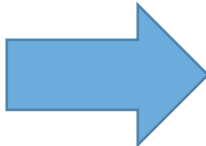
Joins: Example

 $R \bowtie S$

```
SELECT R.A, B, C, D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples $r \in R, s \in S$ such that $r.A = s.A$

R			S					
A	B	C	A	D				
1	0	1	3	7				
2	3	4	2	2				
2	5	2	2	3				
3	1	1						



A	B	C	D
2	3	4	2
2	3	4	3
2	5	2	2

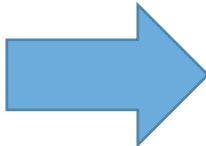
Joins: Example

 $R \bowtie S$

```
SELECT R.A, B, C, D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples $r \in R, s \in S$ such that $r.A = s.A$

R			S					
A	B	C	A	D				
1	0	1	3	7				
2	3	4	2	2				
2	5	2	2	3				
3	1	1						



A	B	C	D
2	3	4	2
2	3	4	3
2	5	2	2
2	5	2	3

Joins: Example

 $R \bowtie S$

```
SELECT R.A, B, C, D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples $r \in R, s \in S$ such that $r.A = s.A$

R

A	B	C
1	0	1
2	3	4
2	5	2
3	1	1

S

A	D
3	7
2	2
2	3

→

A	B	C	D
2	3	4	2
2	3	4	3
2	5	2	2
2	5	2	3
3	1	1	7

Semantically: A Subset of the Cross Product

 $R \bowtie S$

```
SELECT R.A, B, C, D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples $r \in R, s \in S$ such that $r.A = s.A$

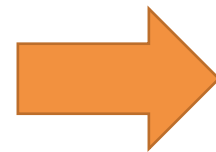
R

A	B	C
1	0	1
2	3	4
2	5	2
3	1	1

×

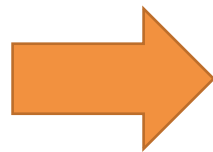
S

A	D
3	7
2	2
2	3



Cross
Product

...



Filter by
conditions
($r.A = s.A$)

A	B	C	D
2	3	4	2
2	3	4	3
2	5	2	2
2	5	2	3
3	1	1	7

Can we actually
implement a join
in this way?

Notes

- We write $\mathbf{R} \bowtie \mathbf{S}$ to mean *join R and S by returning all tuple pairs where **all shared attributes** are equal*
- We write $\mathbf{R} \bowtie \mathbf{S} \text{ on } \mathbf{A}$ to mean *join R and S by returning all tuple pairs where **attribute(s) A** are equal*
- For simplicity, we'll consider joins on **two tables** and with **equality constraints** (“equijoins”)

However joins *can* merge > 2 tables, and some algorithms do support non-equality constraints!

Nested Loop Joins

Notes

- We are again considering “IO aware” algorithms:
care about disk IO
- Given a relation R , let:
 - $T(R)$ = # of tuples in R
 - $P(R)$ = # of pages in R
- Note also that we omit ceilings in calculations...
good exercise to put back in!

Recall that we read / write
entire pages with disk IO

Nested Loop Join (NLJ)

```
Compute  $R \bowtie S$  on  $A$ :  
  for  $r$  in  $R$ :  
    for  $s$  in  $S$ :  
      if  $r[A] == s[A]$ :  
        yield  $(r, s)$ 
```

Nested Loop Join (NLJ)

```
Compute  $R \bowtie S$  on  $A$ :
```

```
  for  $r$  in  $R$ :
```

```
    for  $s$  in  $S$ :
```

```
      if  $r[A] == s[A]$ :
```

```
        yield  $(r, s)$ 
```

Cost:

$P(R)$

1. Loop over the tuples in R

Note that our IO cost is based on the number of *pages* loaded, not the number of tuples!

Nested Loop Join (NLJ)

```
Compute  $R \bowtie S$  on  $A$ :  
  for  $r$  in  $R$ :  
    for  $s$  in  $S$ :  
      if  $r[A] == s[A]$ :  
        yield  $(r, s)$ 
```

Cost:

$$P(R) + T(R) * P(S)$$

1. Loop over the tuples in R
2. For every tuple in R , loop over all the tuples in S

Have to read *all of S* from disk for *every tuple in R !*

Nested Loop Join (NLJ)

```
Compute  $R \bowtie S$  on  $A$ :  
  for  $r$  in  $R$ :  
    for  $s$  in  $S$ :  
      if  $r[A] == s[A]$ :  
        yield  $(r, s)$ 
```

Cost:

$$P(R) + T(R) * P(S)$$

1. Loop over the tuples in R
2. For every tuple in R , loop over all the tuples in S
3. Check against join conditions

Note that NLJ can handle things other than equality constraints... just check in the *if* statement!

Nested Loop Join (NLJ)

```
Compute  $R \bowtie S$  on  $A$ :  
  for  $r$  in  $R$ :  
    for  $s$  in  $S$ :  
      if  $r[A] == s[A]$ :  
        yield ( $r, s$ )
```

What would *OUT* be if our join condition is trivial (if *TRUE*)?

OUT could be bigger than $P(R)*P(S)$... but usually not that bad

Cost:

$$P(R) + T(R)*P(S) + OUT$$

1. Loop over the tuples in R
2. For every tuple in R , loop over all the tuples in S
3. Check against join conditions
4. **Write out (to page, then when page full, to disk)**

Nested Loop Join (NLJ)

```
Compute  $R \bowtie S$  on  $A$ :  
  for  $r$  in  $R$ :  
    for  $s$  in  $S$ :  
      if  $r[A] == s[A]$ :  
        yield  $(r, s)$ 
```

Cost:

$$P(R) + T(R) * P(S) + \text{OUT}$$

What if R ("outer") and S ("inner") switched?



$$P(S) + T(S) * P(R) + \text{OUT}$$

Outer vs. inner selection makes a huge difference-
DBMS needs to know which relation is smaller!

IO-Aware Approach

Block Nested Loop Join (BNLJ)

Given $B+1$ pages of memory

Compute $R \bowtie S$ on A :

for each $B-1$ pages pr of R :

for page ps of S :

for each tuple r in pr :

for each tuple s in ps :

if $r[A] == s[A]$:

yield (r, s)

Cost:

$P(R)$

1. Load in $B-1$ pages of R at a time (leaving 1 page each free for S & output)

Note: There could be some speedup here due to the fact that we're reading in multiple pages sequentially however we'll ignore this here!

Block Nested Loop Join (BNLJ)

Given $B+1$ pages of memory

Compute $R \bowtie S$ on A :

for each $B-1$ pages pr of R :

for page ps of S :

for each tuple r in pr :

for each tuple s in ps :

if $r[A] == s[A]$:

yield (r, s)

Cost:

$$P(R) + P(R)/(B-1) P(S)$$

1. Load in $B-1$ pages of R at a time (leaving 1 page each free for S & output)
2. For each $(B-1)$ -page segment of R , load each page of S

Note: Faster to iterate over the *smaller* relation first!

Block Nested Loop Join (BNLJ)

Given $B+1$ pages of memory

Compute $R \bowtie S$ on A :

```

for each B-1 pages pr of R:
  for page ps of S:
    for each tuple r in pr:
      for each tuple s in ps:
        if r[A] == s[A]:
          yield (r,s)
  
```

Cost:

$$P(R) + P(R)/(B-1) P(S)$$

1. Load in B-1 pages of R at a time (leaving 1 page each free for S & output)
2. For each (B-1)-page segment of R, load each page of S
3. Check against the join conditions

BNLJ can also handle non-equality constraints

Block Nested Loop Join (BNLJ)

Given $B+1$ pages of memory

Compute $R \bowtie S$ on A :

```

for each B-1 pages pr of R:
  for page ps of S:
    for each tuple r in pr:
      for each tuple s in ps:
        if r[A] == s[A]:
          yield (r,s)
  
```

Again, OUT could be bigger than $P(R)*P(S)$... but usually not that bad

Cost:

$$P(R) + P(R)/B - 1 P(S) + OUT$$

1. Load in B-1 pages of R at a time (leaving 1 page each free for S & output)
2. For each (B-1)-page segment of R, load each page of S
3. Check against the join conditions

4. Write out

BNLJ vs. NLJ: Benefits of IO Aware

- In BNLJ, by loading larger chunks of R, we minimize the number of full *disk reads* of S
 - We only read all of S from disk for ***every (B-1)-page segment of R!***
 - Still the full cross-product, but more done only *in memory*

NLJ

$$P(R) + T(R) * P(S) + \text{OUT}$$



BNLJ

$$P(R) + \frac{P(R)}{B-1} P(S) + \text{OUT}$$

BNLJ is faster by roughly $\frac{(B-1)T(R)}{P(R)}$!

BNLJ vs. NLJ: Benefits of IO Aware

- Example:
 - R: 500 pages
 - S: 1000 pages
 - 100 tuples / page
 - We have 12 pages of memory ($B = 11$)
- NLJ: Cost = $500 + 50,000 * 1000 = 50 \text{ Million IOs} \approx \underline{140 \text{ hours}}$
- BNLJ: Cost = $500 + 500 * 1000 / 10 = 50 \text{ Thousand IOs} \approx \underline{0.14 \text{ hours}}$

Ignoring OUT here...

A very real difference from a small
change in the algorithm!

Smarter than Cross-Products

Smarter than Cross-Products: From Quadratic to Nearly Linear

- All joins that compute the **full cross-product** have some **quadratic** term

• For example we saw: NLJ $P(R) + \mathbf{T(R)P(S)} + \text{OUT}$

BNLJ $P(R) + \mathbf{P(R)/B-1 P(S)} + \text{OUT}$

- Now we'll see some (nearly) linear joins:
 - $\sim O(P(R) + P(S) + \mathbf{OUT})$, where again **OUT** could be quadratic but is usually better

We get this gain by *taking advantage of structure*- moving to equality constraints ("equijoin") only!

Index Nested Loop Join (INLJ)

Compute $R \bowtie S$ on A :

Given index idx on $S.A$:

for r in R :

s in $idx(r[A])$:

 yield r, s

Cost:

$$P(R) + T(R) * L + OUT$$

where L is the IO cost to access all the distinct values in the index; assuming these fit on one page, $L \sim 3$ is good est.

→ We can use an index (e.g. B+ Tree) to *avoid doing the full cross-product!*

Joins: A Cage Match

Message: It's all about the memory!

Today's Lecture

1. Sort-Merge Join (SMJ)
2. Hash Join (HJ)
3. The Cage Match: SMJ vs. HJ

1. Sort-Merge Join (SMJ)



What you will learn about in this section

1. Sort-Merge Join
2. “Backup” & Total Cost
3. Optimizations
4. ACTIVITY: Sequential Flooding

Sort Merge Join (SMJ): Basic Procedure

To compute $R \bowtie S$ on A :

Note that we are only considering equality join conditions here

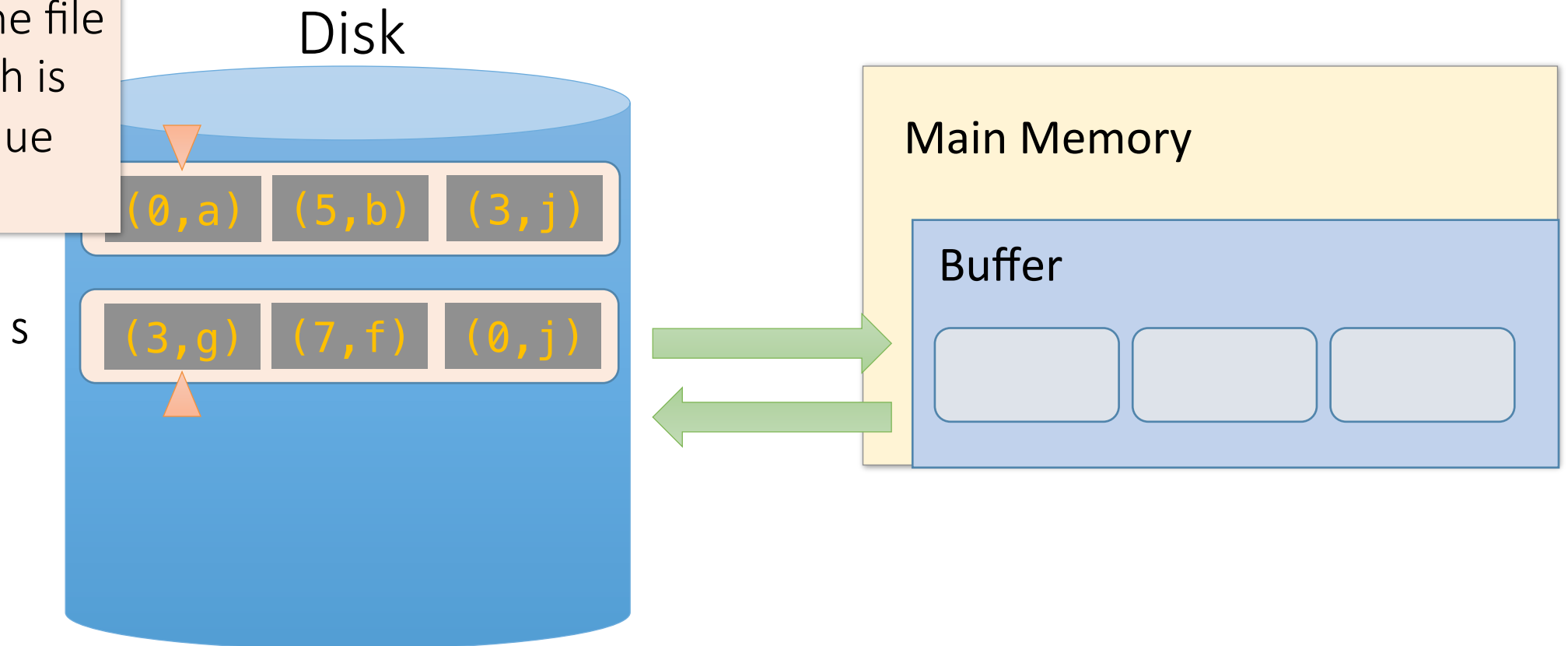
1. Sort R, S on A using ***external merge sort***
2. ***Scan*** sorted files and “merge”
3. *[May need to “backup”- see next subsection]*

Note that if R, S are already sorted on A ,
SMJ will be awesome!

SMJ Example: $R \bowtie S$ on A with 3 page buffer

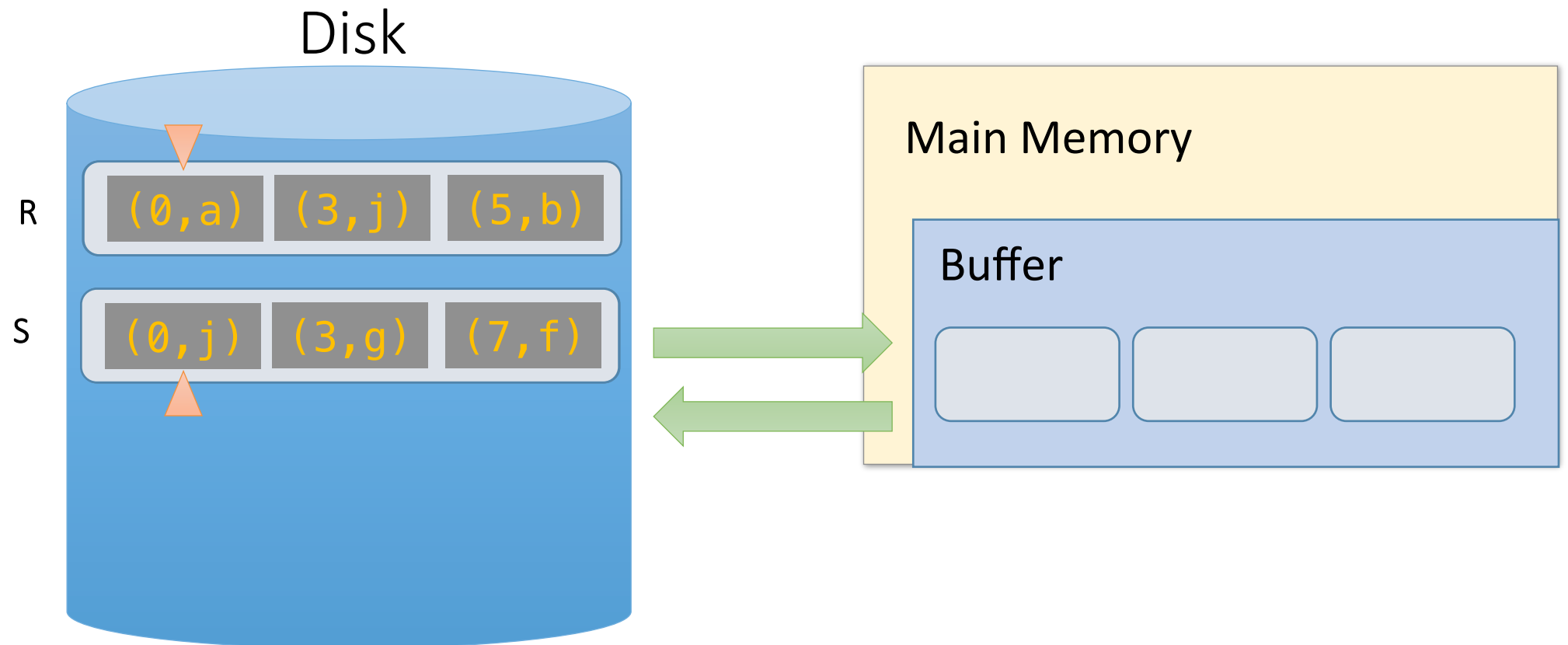
- For simplicity: Let each page be **one tuple**, and let the first value be A

We show the file HEAD, which is the next value to be read!



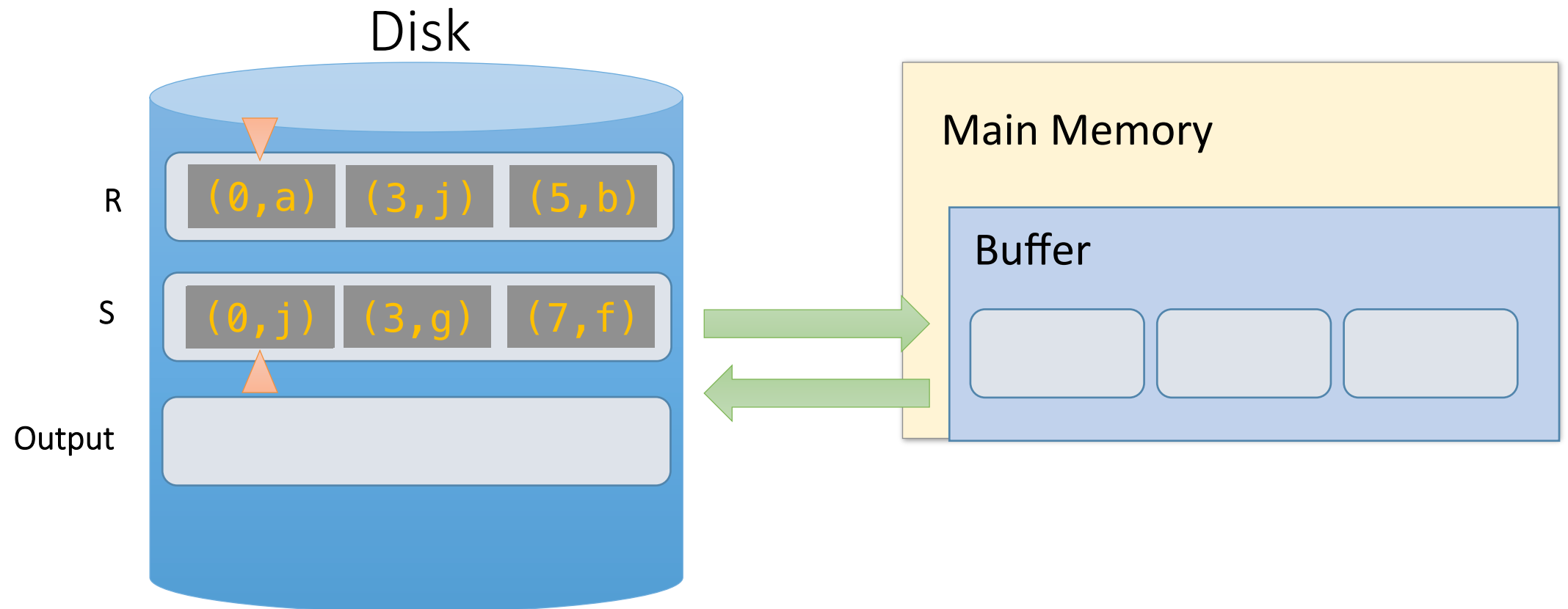
SMJ Example: $R \bowtie S$ on A with 3 page buffer

1. Sort the relations R, S on the join key (first value)



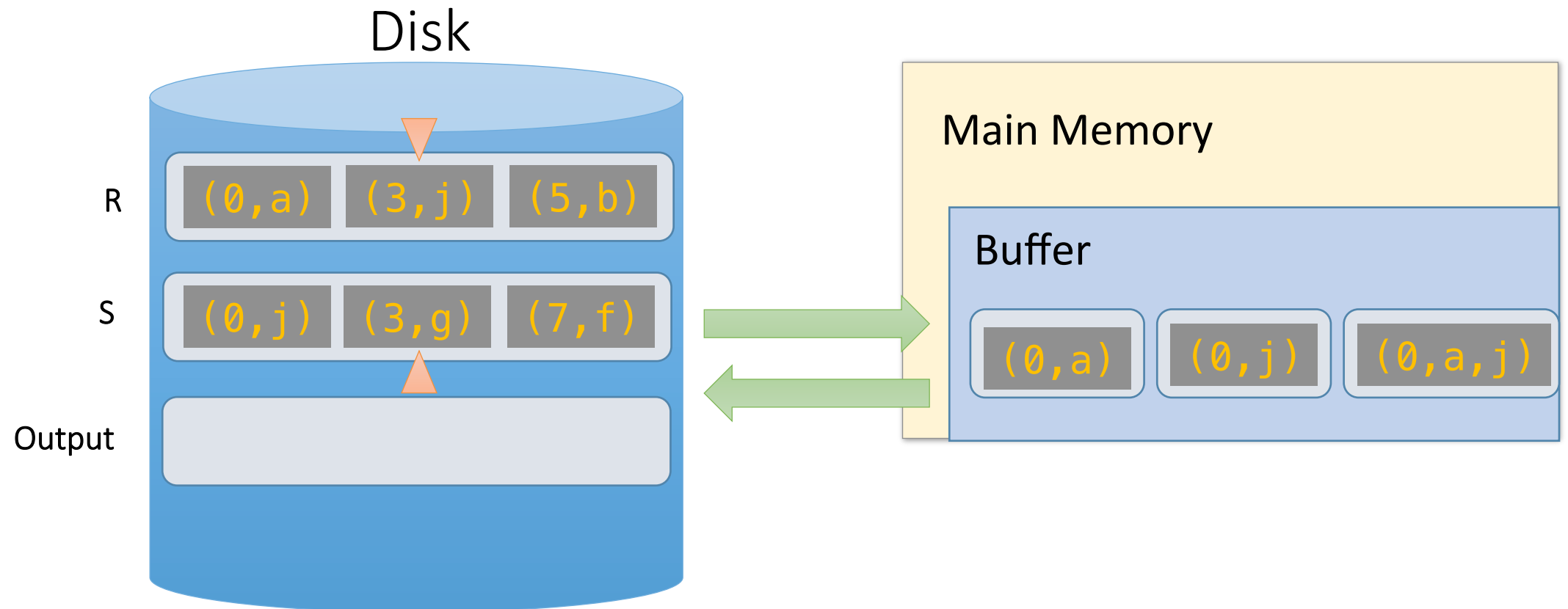
SMJ Example: $R \bowtie S$ on A with 3 page buffer

2. Scan and “merge” on join key!



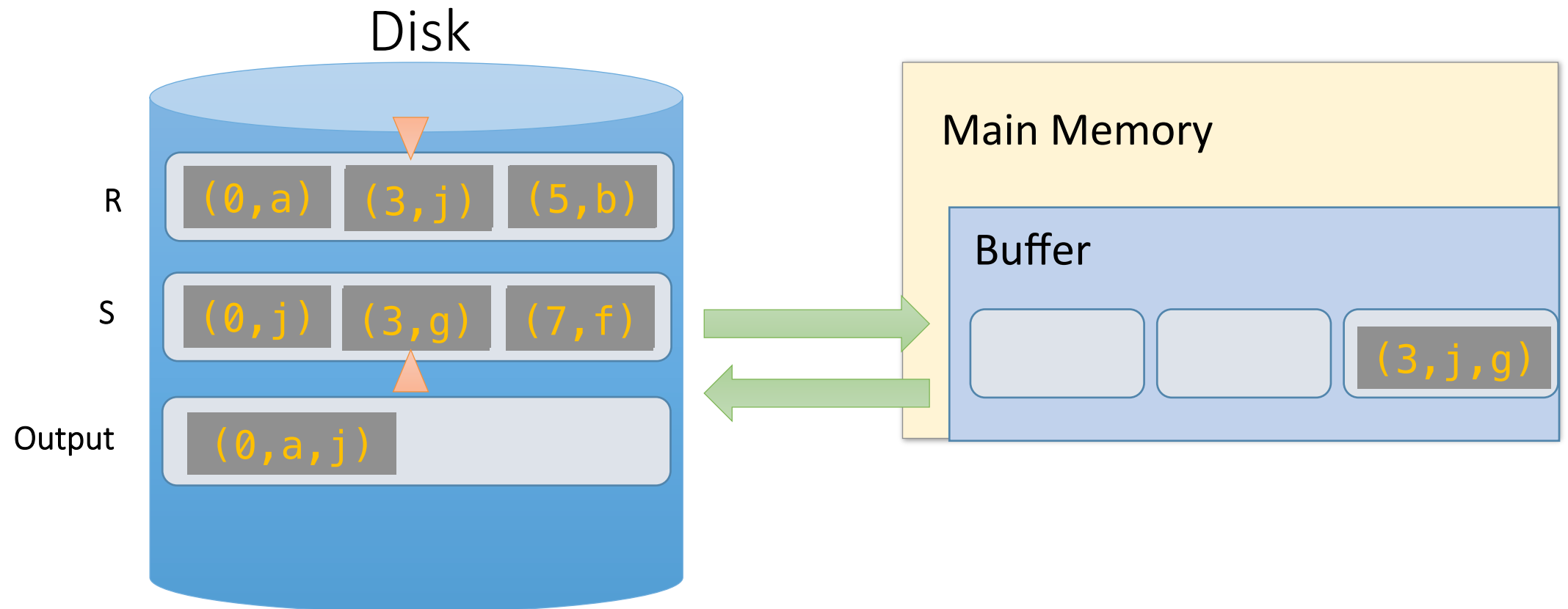
SMJ Example: $R \bowtie S$ on A with 3 page buffer

2. Scan and “merge” on join key!



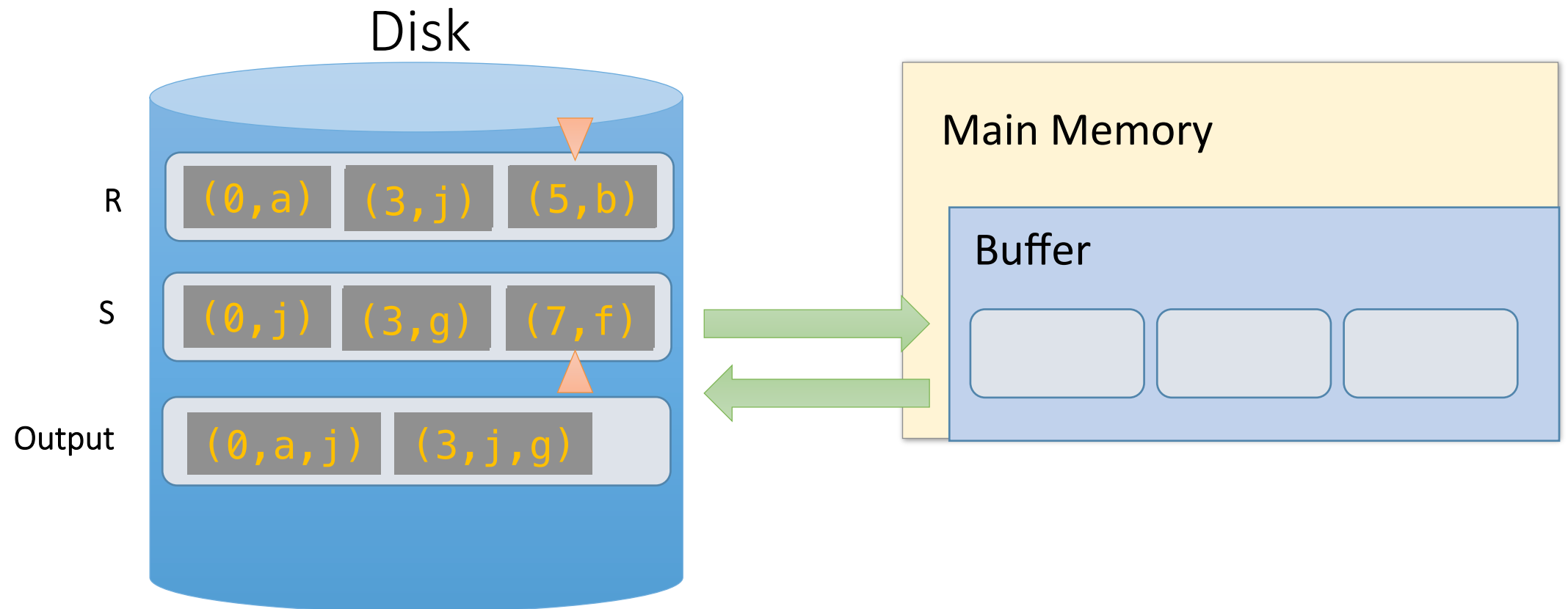
SMJ Example: $R \bowtie S$ on A with 3 page buffer

2. Scan and “merge” on join key!



SMJ Example: $R \bowtie S$ on A with 3 page buffer

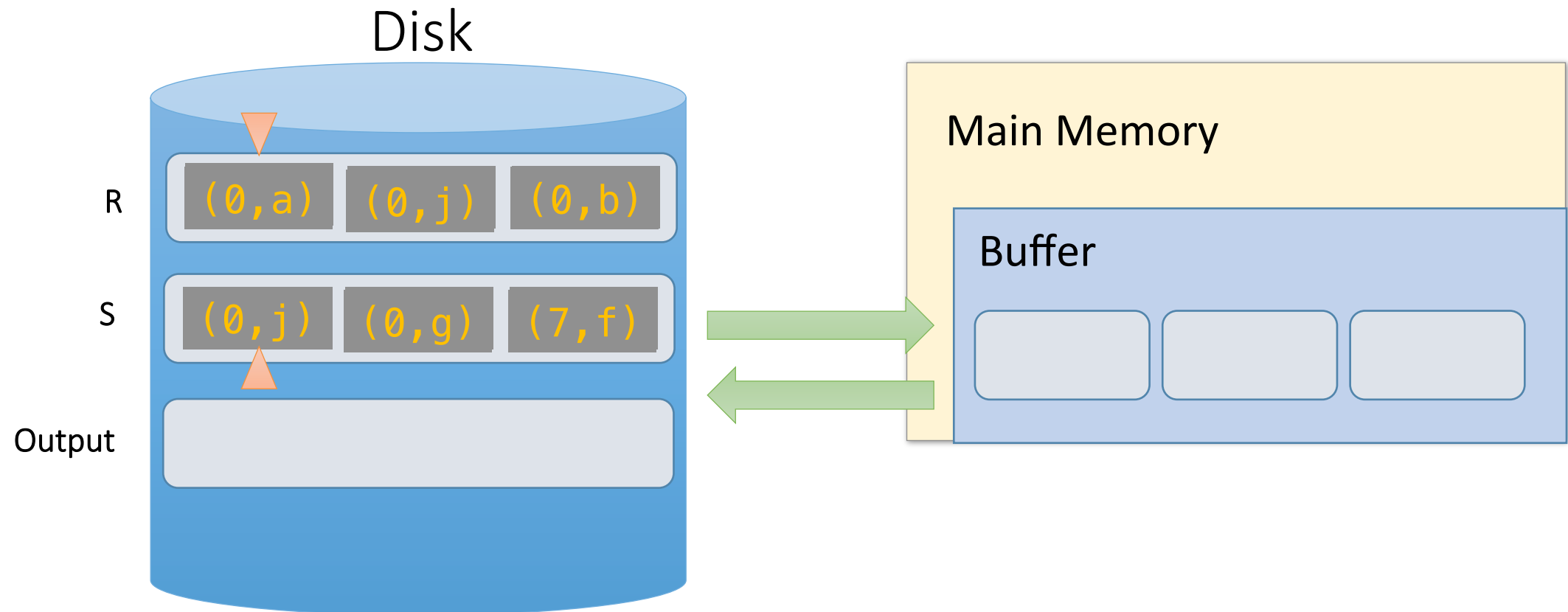
2. Done!



What happens with duplicate join keys?

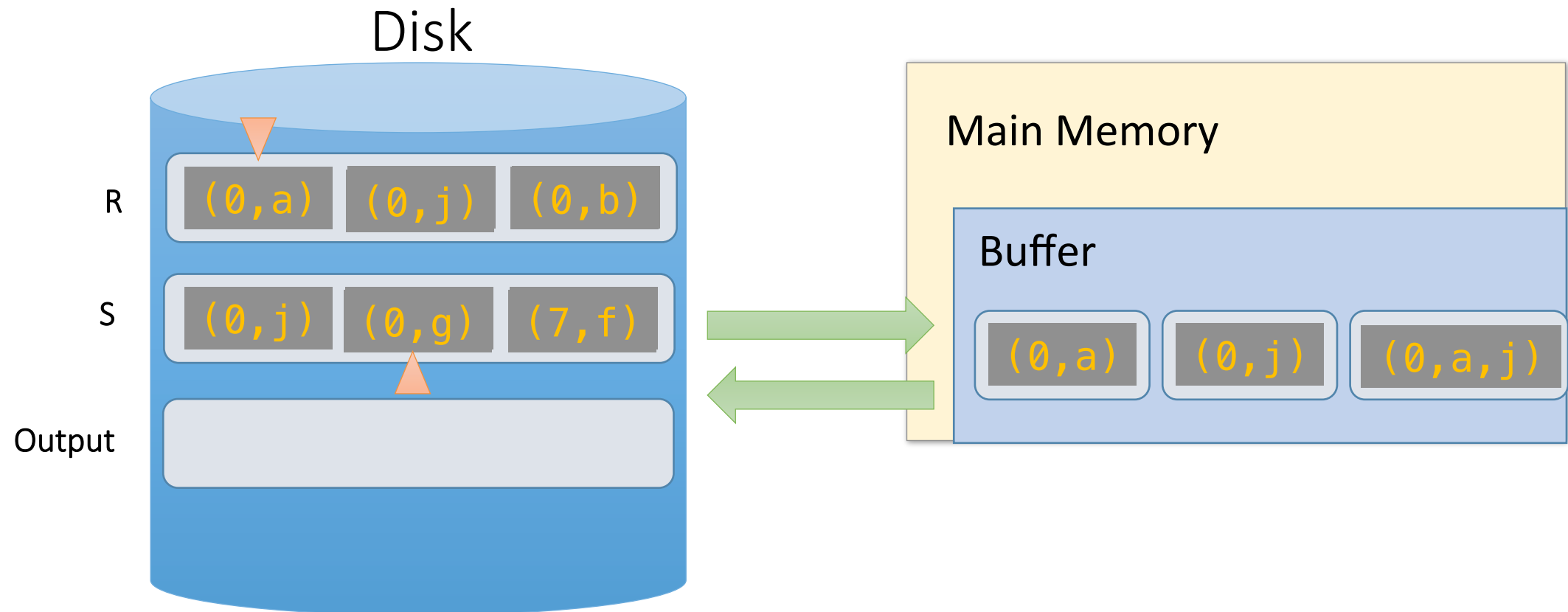
Multiple tuples with Same Join Key: “Backup”

1. Start with sorted relations, and begin scan / merge...



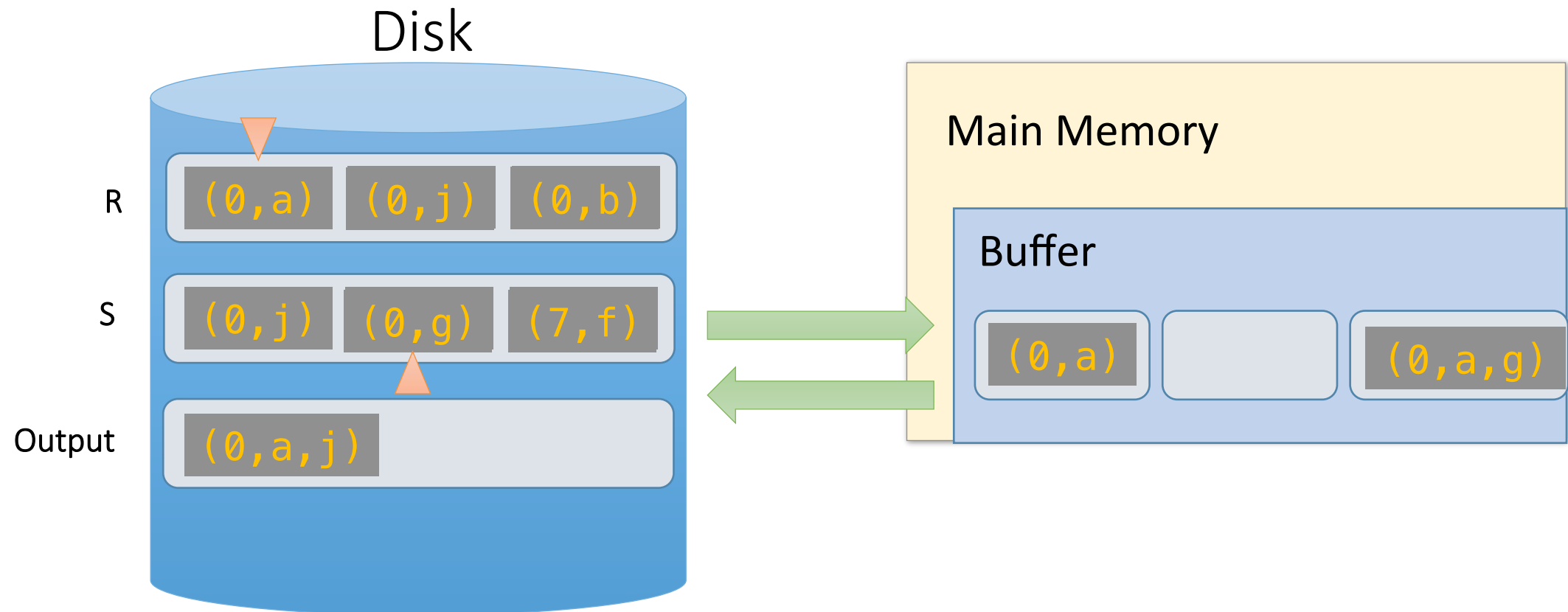
Multiple tuples with Same Join Key: “Backup”

1. Start with sorted relations, and begin scan / merge...



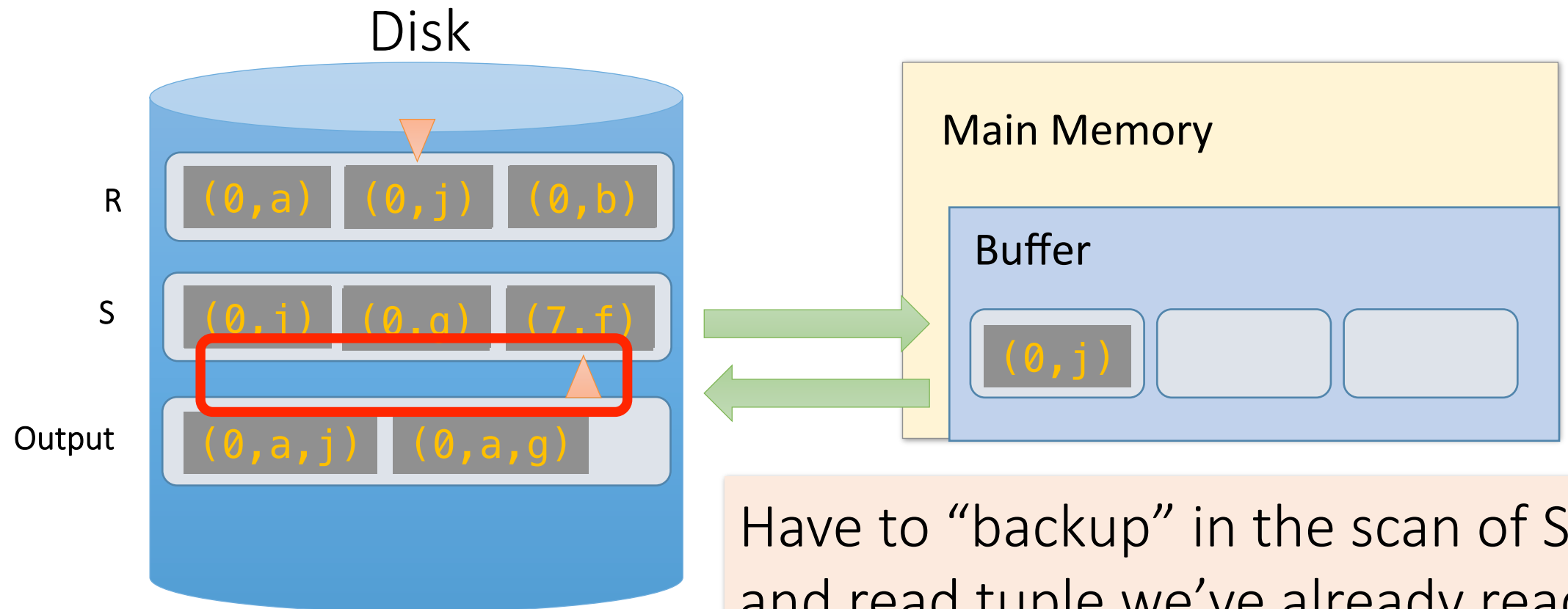
Multiple tuples with Same Join Key: “Backup”

1. Start with sorted relations, and begin scan / merge...



Multiple tuples with Same Join Key: “Backup”

1. Start with sorted relations, and begin scan / merge...



Backup

- At best, no backup $\rightarrow$ scan takes **$P(R) + P(S)$** reads
 - For ex: if no duplicate values in join attribute
- At worst (e.g. full backup each time), scan could take **$P(R) * P(S)$** reads!
 - For ex: if *all* duplicate values in join attribute, i.e. all tuples in R and S have the same value for the join attribute
 - Roughly: For each page of R, we'll have to *back up* and read each page of S...
- Often not that bad however, plus we can:
 - Leave more data in buffer (for larger buffers)
 - Can “zig-zag” (see animation)

SMJ: Total cost

- Cost of SMJ is **cost of sorting** R and S...
- Plus the **cost of scanning**: $\sim P(R) + P(S)$
 - Because of *backup*: in worst case $P(R) * P(S)$; but this would be very unlikely
- Plus the **cost of writing out**: $\sim P(R) + P(S)$ but in worst case $T(R) * T(S)$

$$\sim \text{Sort}(P(R)) + \text{Sort}(P(S)) \\ + P(R) + P(S) + \text{OUT}$$

Recall: $\text{Sort}(N) \approx 2N(\lceil \log_B N \rceil / 2(B+1) + 1)$

Note: this is using repacking, where we estimate that we can create initial runs of length $\sim 2(B+1)$

SMJ vs. BNLJ: Steel Cage Match

- If we have 100 buffer pages, $P(R) = 1000$ pages and $P(S) = 500$ pages:
 - Sort both in two passes: $2 * 2 * 1000 + 2 * 2 * 500 = \mathbf{6,000 \text{ IOs}}$
 - Merge phase $1000 + 500 = 1,500 \text{ IOs}$
 - **$= 7,500 \text{ IOs} + \text{OUT}$**

What is BNLJ?

- $500 + 1000 * \lceil 500 / 98 \rceil = \mathbf{6,500 \text{ IOs} + \text{OUT}}$
- But, if we have 35 buffer pages?
 - Sort Merge has same behavior (still 2 passes)
 - BNLJ? **$15,500 \text{ IOs} + \text{OUT!}$**



SMJ is $\sim$ linear vs. BNLJ is quadratic...
But it's all about the memory.

A Simple Optimization: Merges Merged!

Given $B+1$ buffer pages

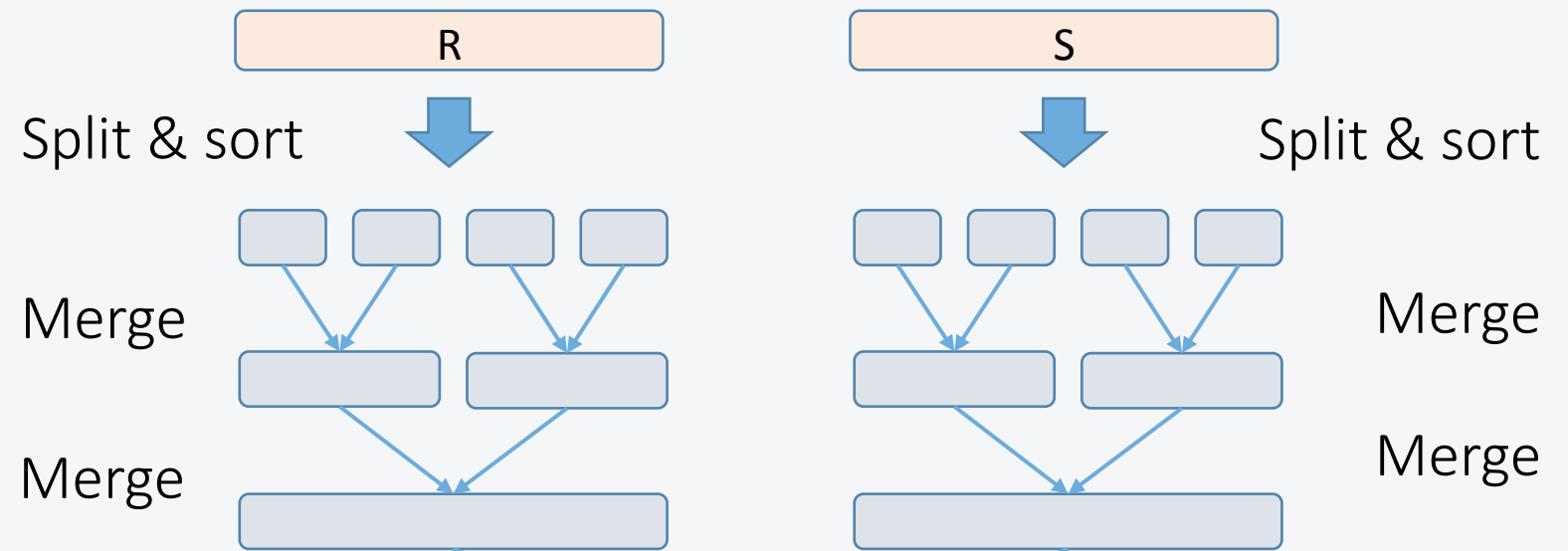
- SMJ is composed of a **sort phase** and a **merge phase**
- During the **sort phase**, run passes of external merge sort on R and S
 - Suppose at some point, R and S have $\leq B$ (sorted) runs in total
 - We could do two merges (for each of R & S) at this point, complete the sort phase, and start the merge phase...
 - OR, we could combine them: do **one** B-way merge and complete the join!

Un-Optimized SMJ

Given $B+1$ buffer pages

Unsorted input relations

Sort Phase
(Ext. Merge Sort)



Merge / Join Phase

Joined output
file created!

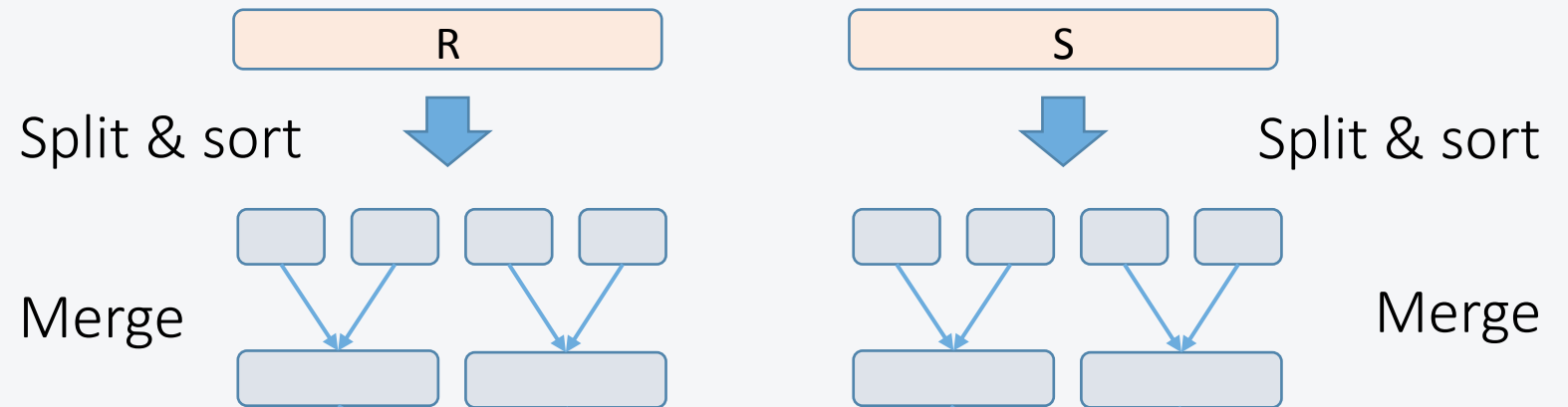
Simple SMJ Optimization

Given $B+1$ buffer pages

Unsorted input relations

Sort Phase
(Ext. Merge Sort)

$\leq B$ total runs



Merge / Join Phase

B-Way Merge / Join

Joined output
file created!

Simple SMJ Optimization

Given $B+1$ buffer pages

- Now, on this last pass, we only do $P(R) + P(S)$ IOs to complete the join!
- If we can initially split R and S into **B total runs each of length approx. $\leq 2(B+1)$** , assuming repacking lets us create initial runs of $\sim 2(B+1)$ - then we only need **$3(P(R) + P(S)) + OUT$** for SMJ!
 - 2 R/W per page to sort runs in memory, 1 R per page to B-way merge / join!
- How much memory for this to happen?
 - $P(R) + P(S) / B \leq 2(B+1) \Rightarrow P(R) + P(S) \leq 2B^2$
 - Thus, **$\max\{P(R), P(S)\} \leq B^2$** is an approximate sufficient condition

See Lecture 15,
Slide 13-14 – to
clarify this slide.

If the larger of R, S has $\leq B^2$ pages, then SMJ costs
 $3(P(R) + P(S)) + OUT$!

Takeaway points from SMJ

If input already sorted on join key, skip the sorts.

- SMJ is basically linear.
- Nasty but unlikely case: Many duplicate join keys.

SMJ needs to sort **both** relations

- If $\max \{ P(R), P(S) \} < B^2$ then cost is $3(P(R)+P(S)) + OUT$

4. Hash Join (HJ)



What you will learn about in this section

1. Hash Join
2. Memory requirements

Recall: Hashing

- **Magic of hashing:**
 - A hash function h_B maps into $[0, B-1]$
 - And maps nearly uniformly
- A hash **collision** is when $x \neq y$ but $h_B(x) = h_B(y)$
 - Note however that it will never occur that $x = y$ but $h_B(x) \neq h_B(y)$
- We hash on an attribute A , so our hash function is $h_B(t)$ has the form $h_B(t.A)$.
 - **Collisions** may be more frequent.

Recall: Mad Hash Collisions



Say something here to justify this slide's existence? [TODO]

Hash Join: High-level procedure

To compute $R \bowtie S$ on A :

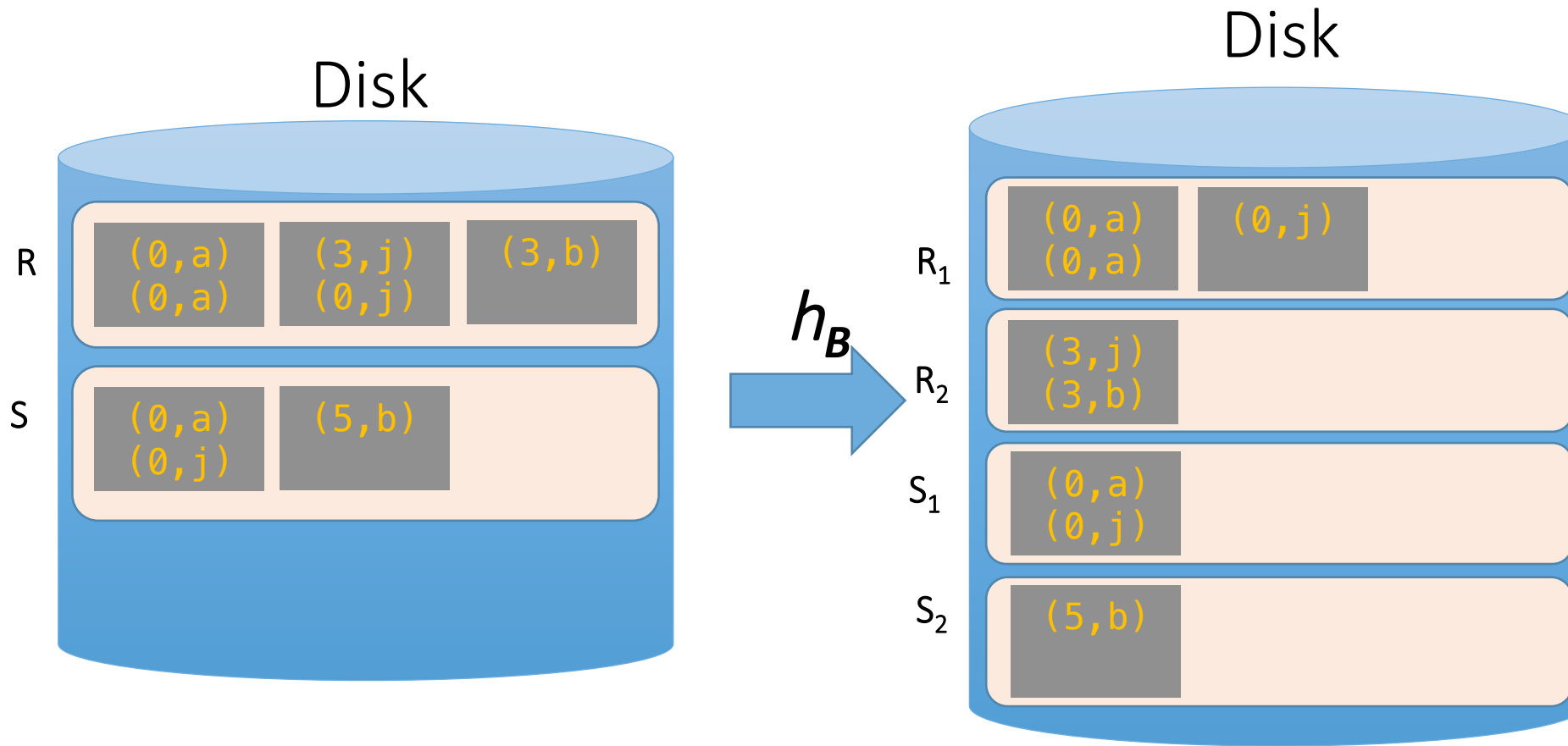
Note again that we are only considering equality constraints here

1. **Partition Phase:** Using one (shared) hash function h_B , partition R and S into B buckets
2. **Matching Phase:** Take pairs of buckets whose tuples have the same values for h , and join these
 1. Use BNLJ here; or hash again $\rightarrow$ either way, operating on small partitions so fast!

We *decompose* the problem using h_B , then complete the join

Hash Join: High-level procedure

1. Partition Phase: Using one (shared) hash function h_B , partition R and S into B buckets

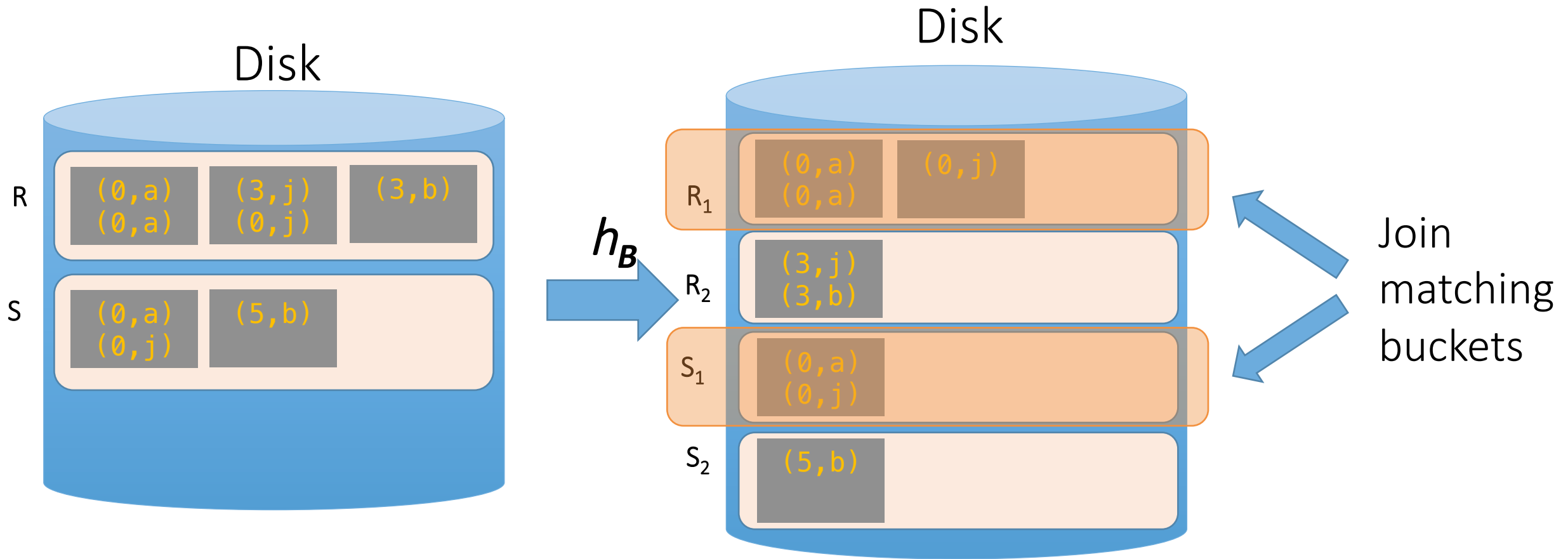


*Note our new convention:
pages each
have two tuples
(one per row)*

More detail in a second...

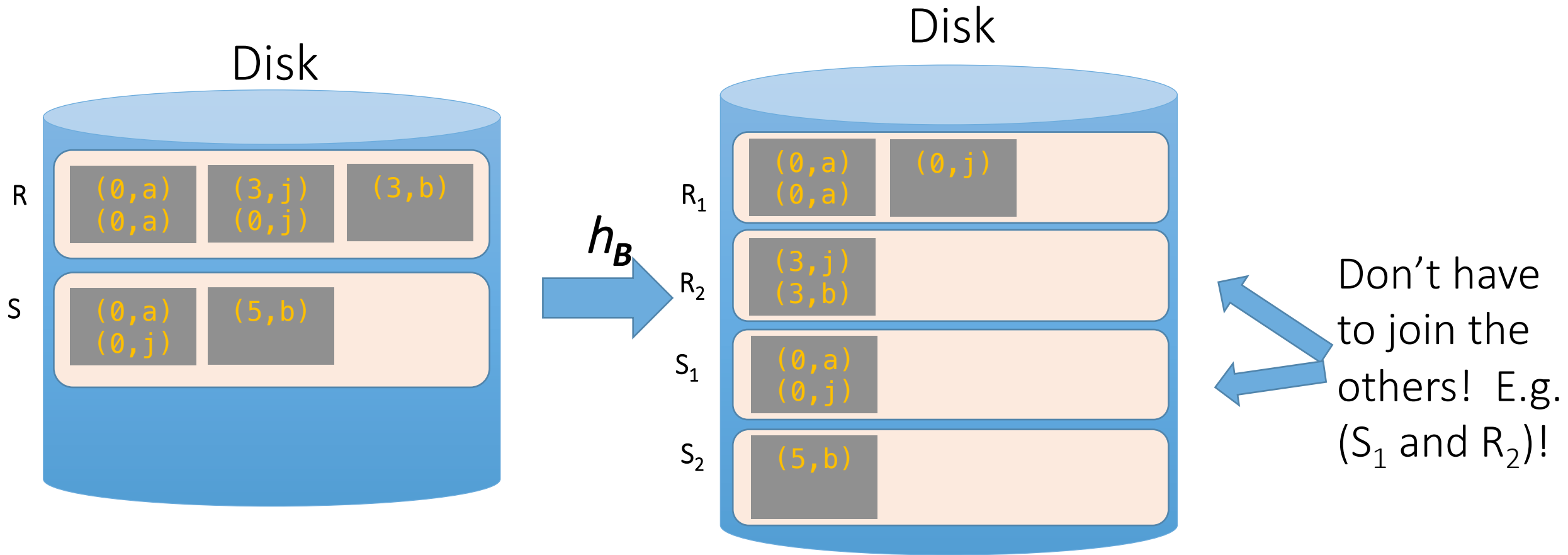
Hash Join: High-level procedure

2. Matching Phase: Take pairs of buckets whose tuples have the same values for h_B , and join these



Hash Join: High-level procedure

2. Matching Phase: Take pairs of buckets whose tuples have the same values for h_B , and join these



Hash Join Phase 1: Partitioning

Goal: For each relation, partition relation into **buckets** such that if $h_B(t.A) = h_B(t'.A)$ they are in the same bucket

Given $B+1$ buffer pages, we partition into B buckets:

- We use B buffer pages for output (one for each bucket), and 1 for input
 - The “dual” of sorting.
 - For each tuple t in input, copy to buffer page for $h_B(t.A)$
 - When page fills up, flush to disk.

How big are the resulting buckets?

Given $B+1$ buffer pages

- Given **N input pages, we partition into B buckets:**
 - → Ideally our buckets are each of size $\sim N/B$ pages
- What happens if there are **hash collisions**?
 - Buckets could be $> N/B$
 - **We'll do several passes...**
- What happens if there are **duplicate join keys**?
 - Nothing we can do here... could have some **skew** in size of the buckets

How big *do we want* the resulting buckets?

Given $B+1$ buffer pages

- Ideally, our buckets would be of size $\leq \mathbf{B-1}$ pages
 - $\mathbf{1}$ for input page, $\mathbf{1}$ for output page, $\mathbf{B-1}$ for each bucket
- Recall: If we want to join a bucket from R and one from S , we can do BNLJ **in linear time** if for *one of them* (wlog say R), $\mathbf{P(R) \leq B-1!}$
 - And more generally, being able to fit bucket in memory is advantageous
- We can keep partitioning buckets that are $> B-1$ pages, until they are $\leq \mathbf{B-1}$ pages
 - Using a new hash key which will split them...

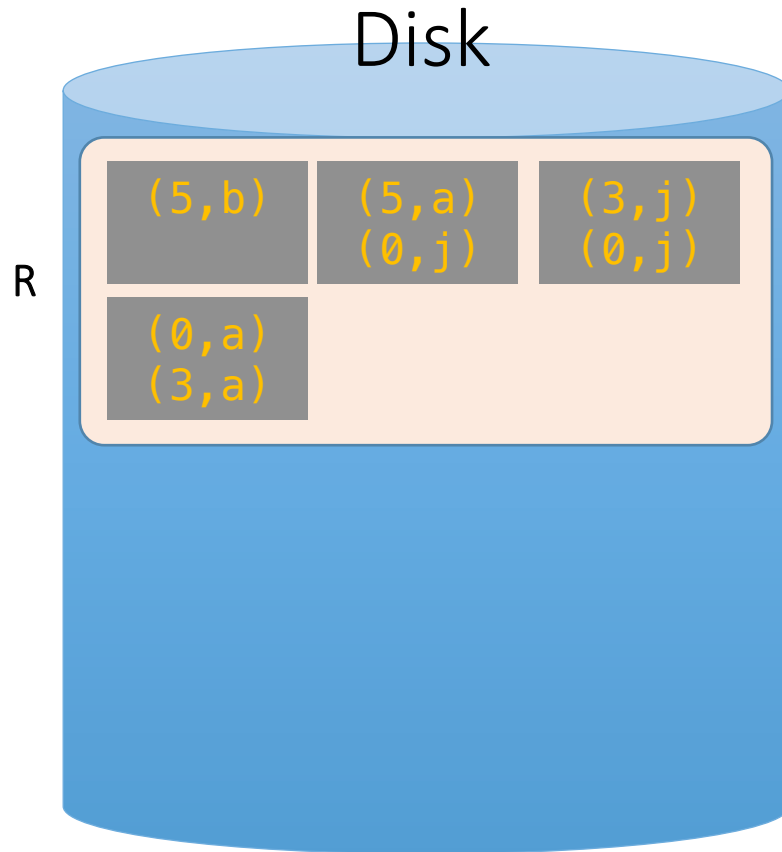
Recall for BNLJ:
 $P(R) + P(R)P(S)/B-1$

We'll call each of these a "pass" again...

Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

We partition into $B = 2$ buckets using hash function h_2 so that we can have one buffer page for each partition (and one for input)



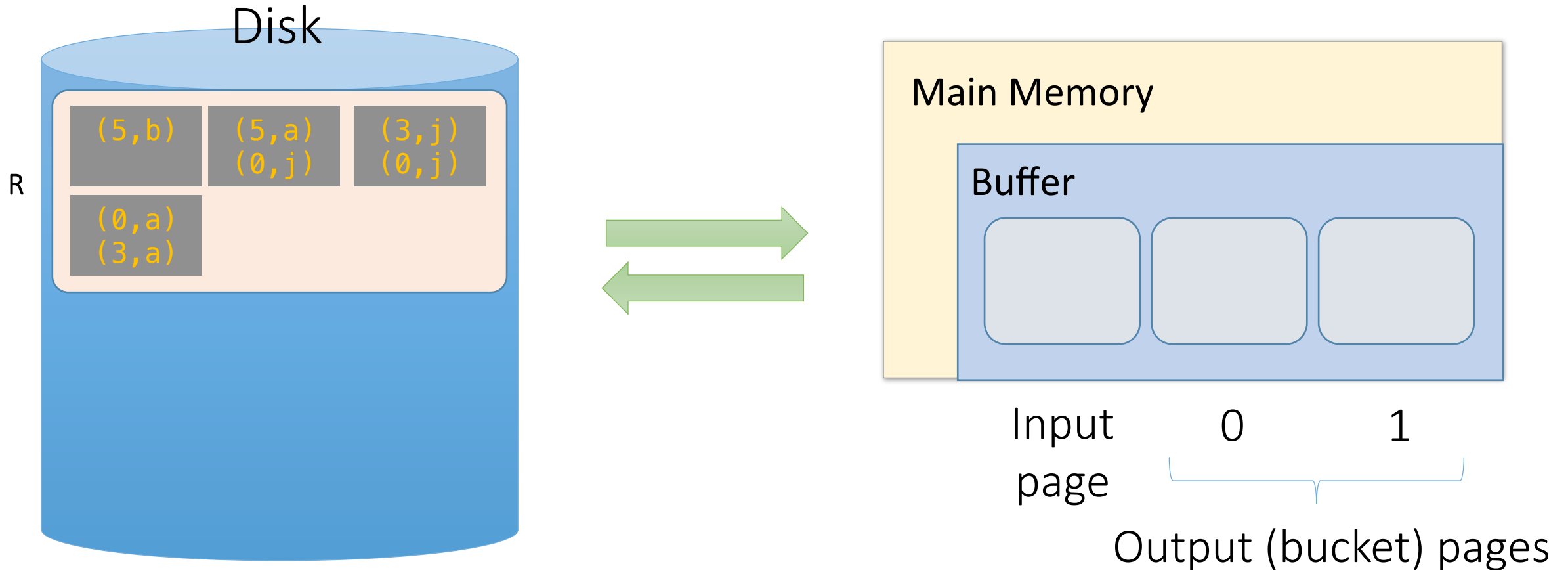
For simplicity, we'll look at partitioning one of the two relations- we just do the same for the other relation!

Recall: our goal will be to get $B = 2$ buckets of size $\leq B-1 \rightarrow 1$ page each

Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

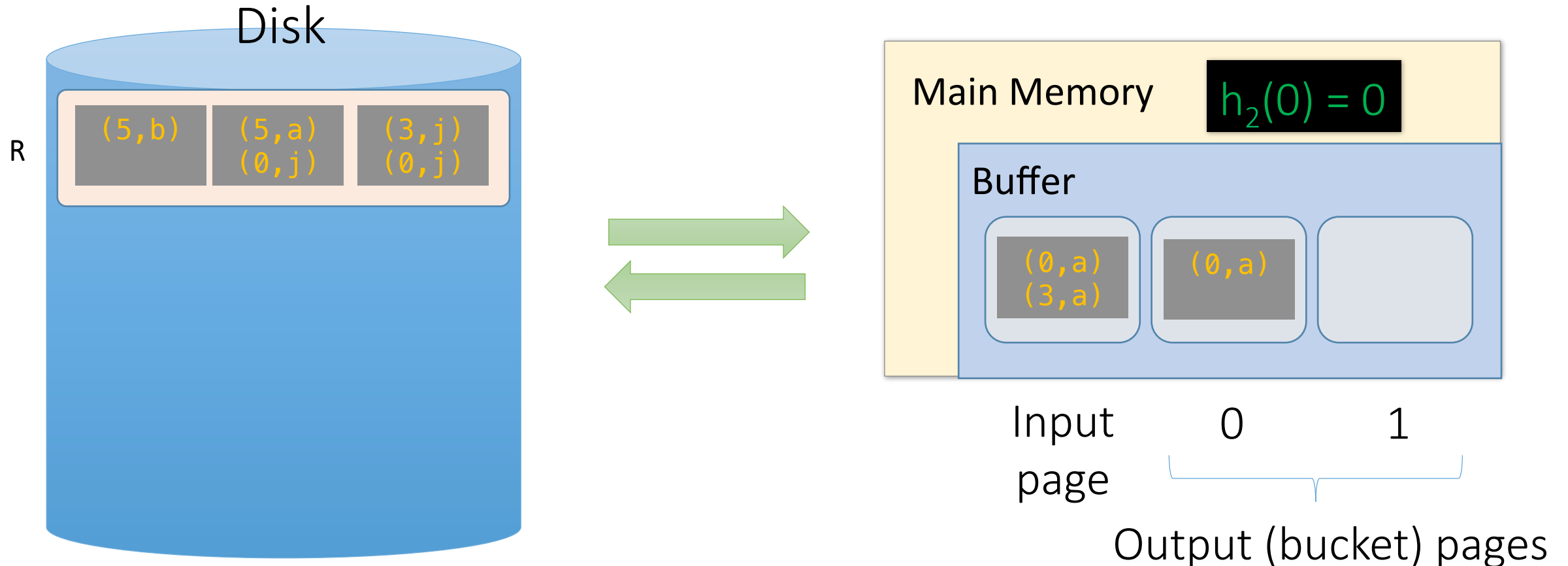
1. We read pages from R into the “input” page of the buffer...



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

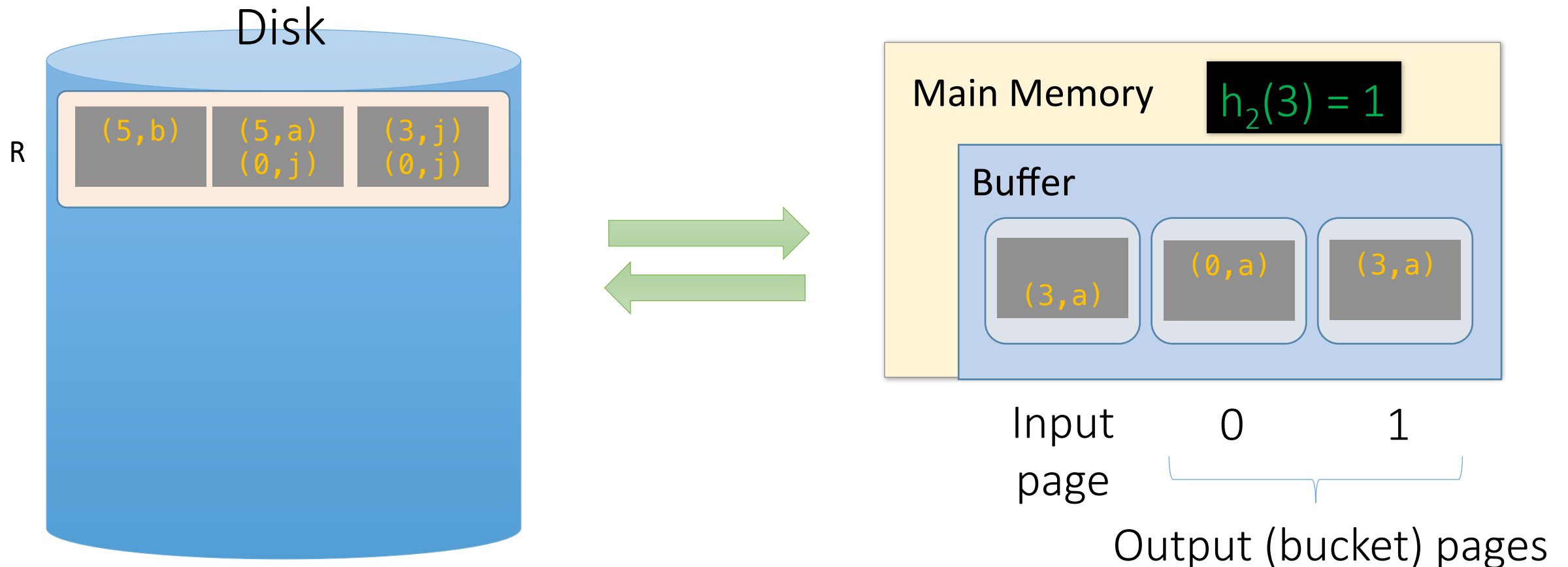
2. Then we use **hash function h_2** to sort into the buckets, which each have one page in the buffer



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

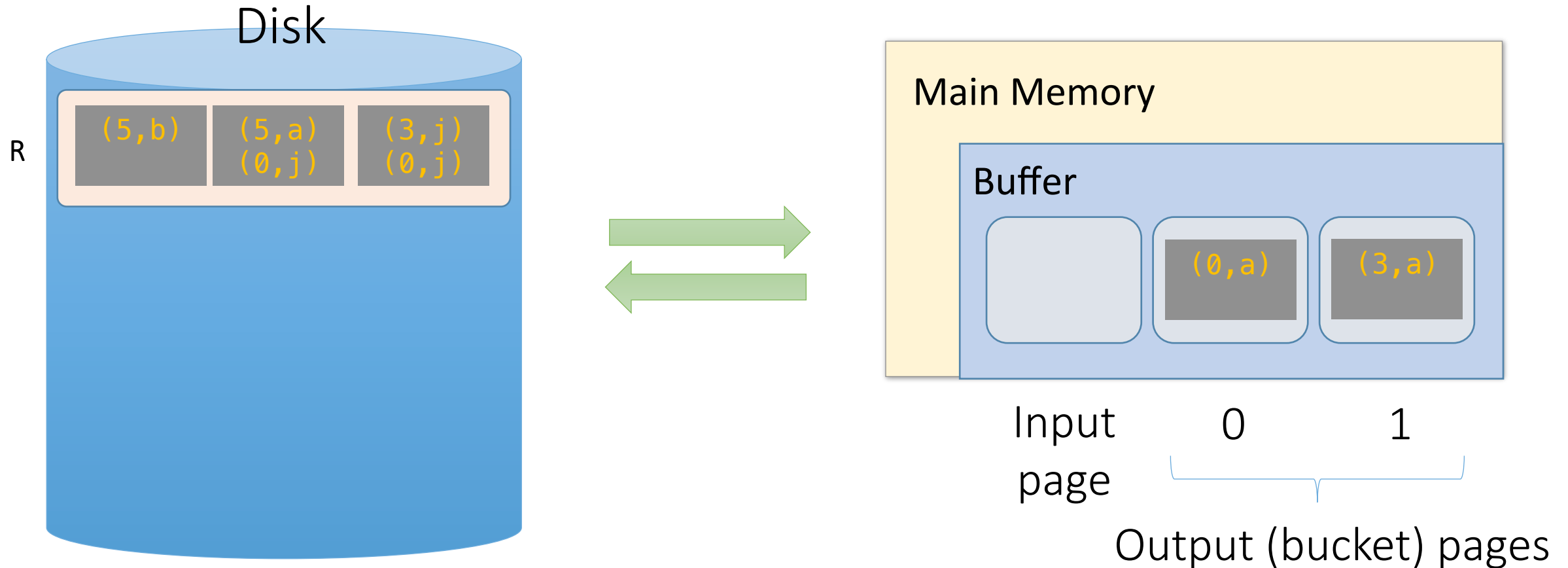
2. Then we use **hash function h_2** to sort into the buckets, which each have one page in the buffer



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

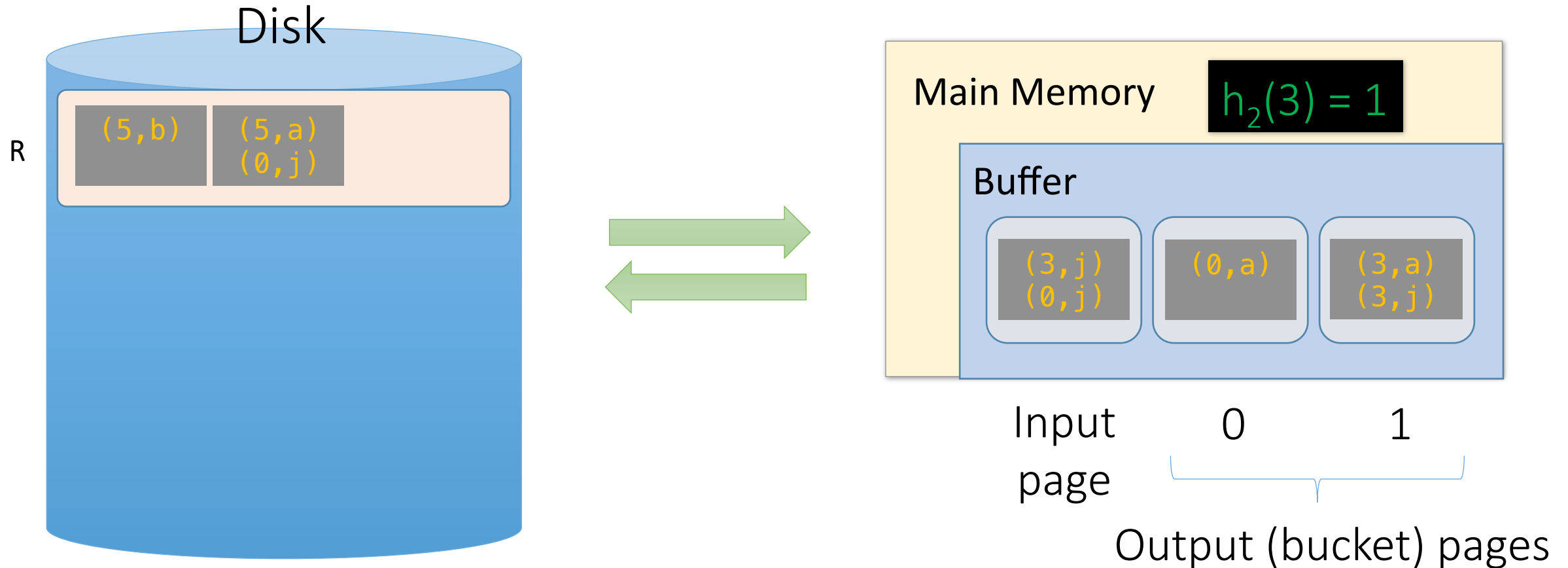
3. We repeat until the buffer bucket pages are full...



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

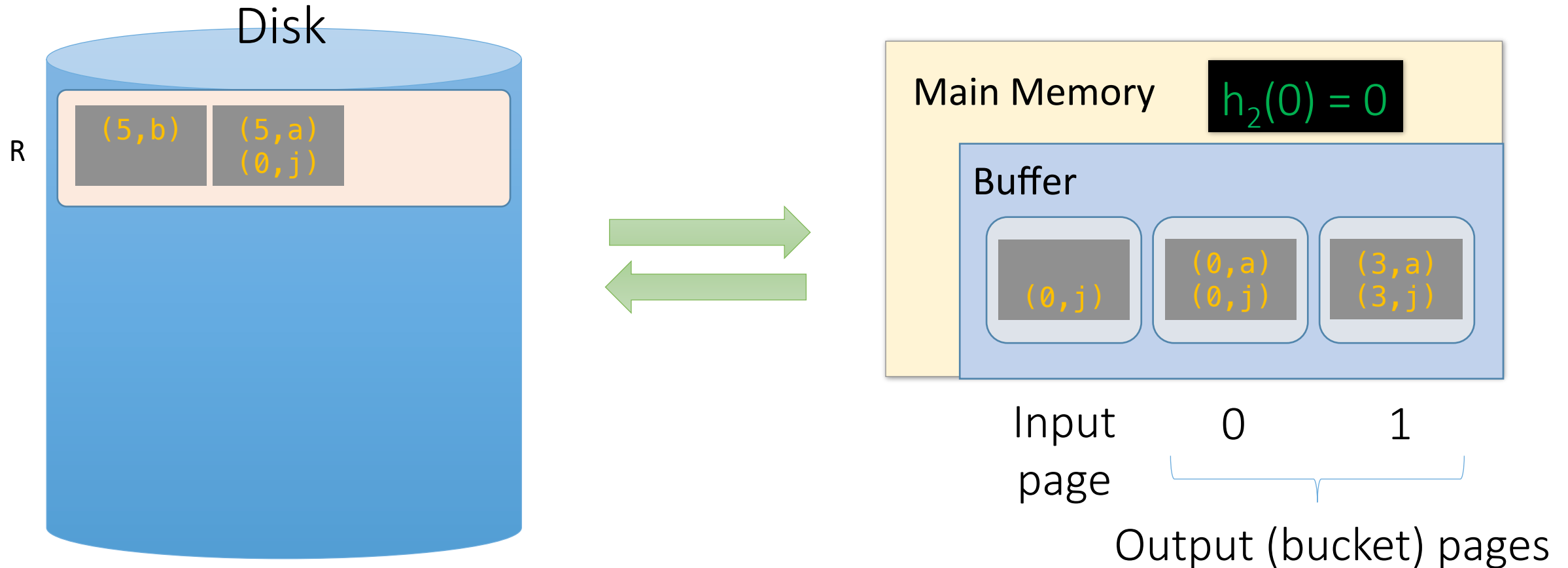
3. We repeat until the buffer bucket pages are full...



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

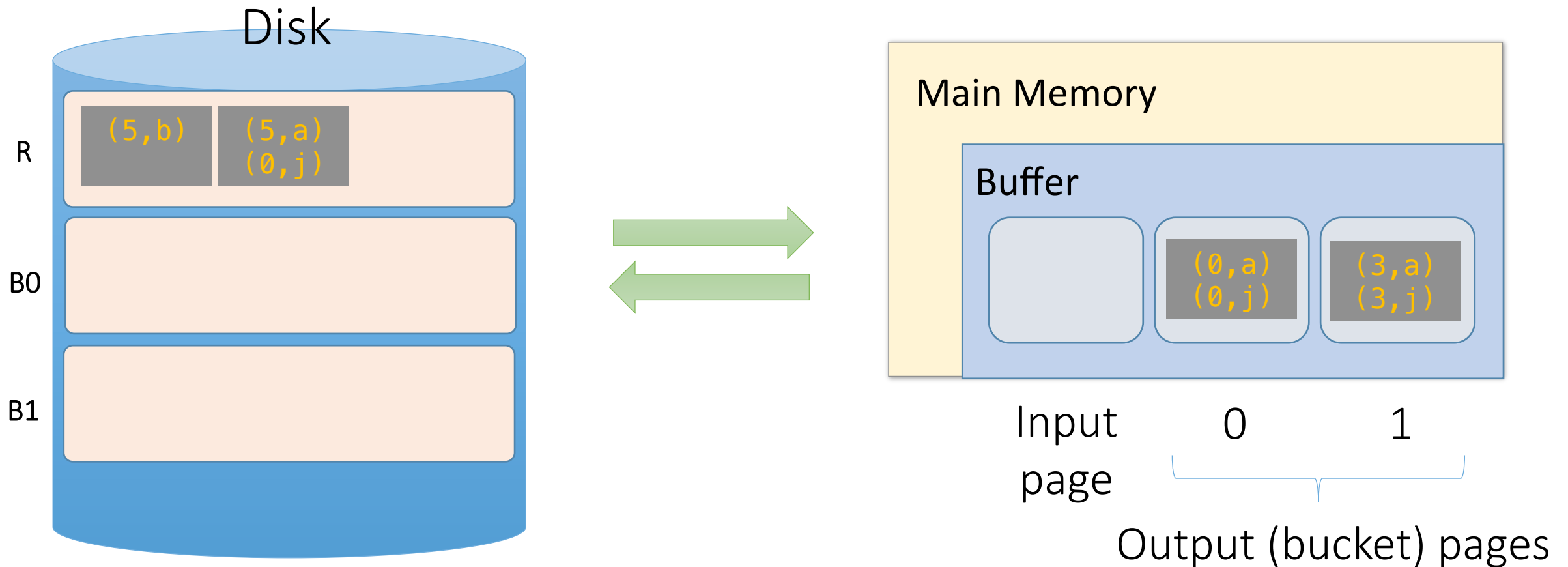
3. We repeat until the buffer bucket pages are full...



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

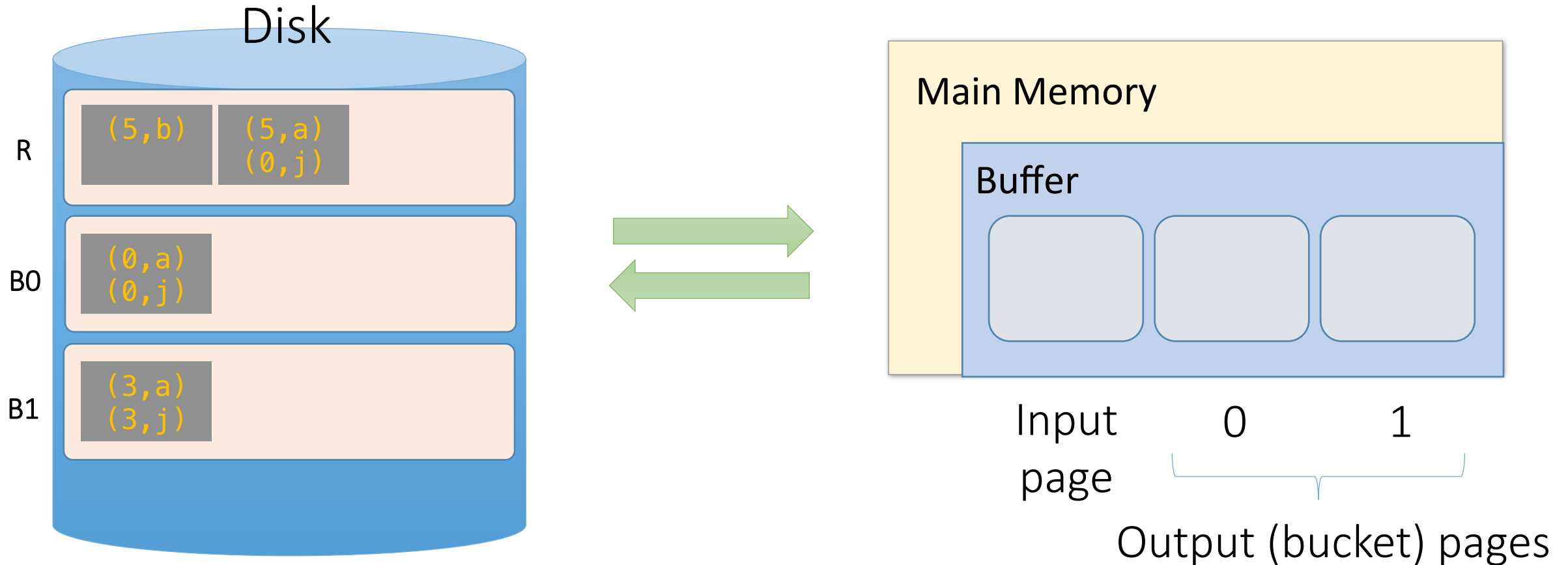
3. We repeat until the buffer bucket pages are full... then flush to disk



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

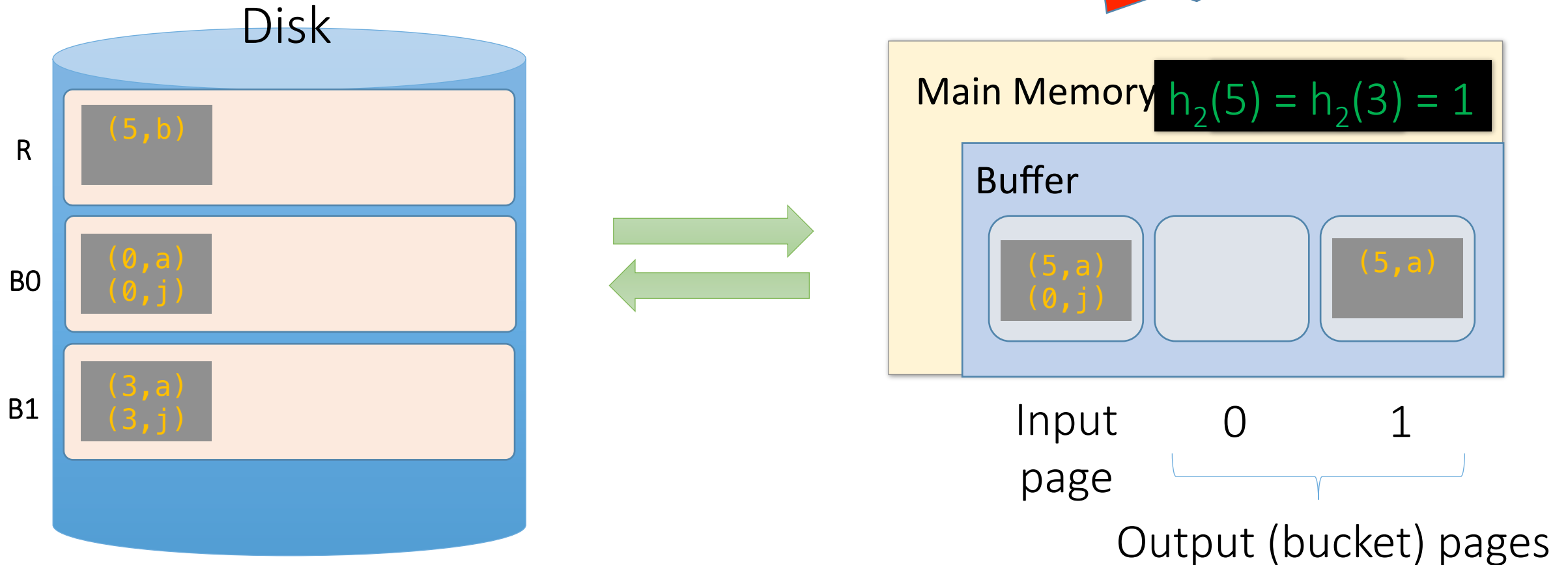
3. We repeat until the buffer bucket pages are full... then flush to disk



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

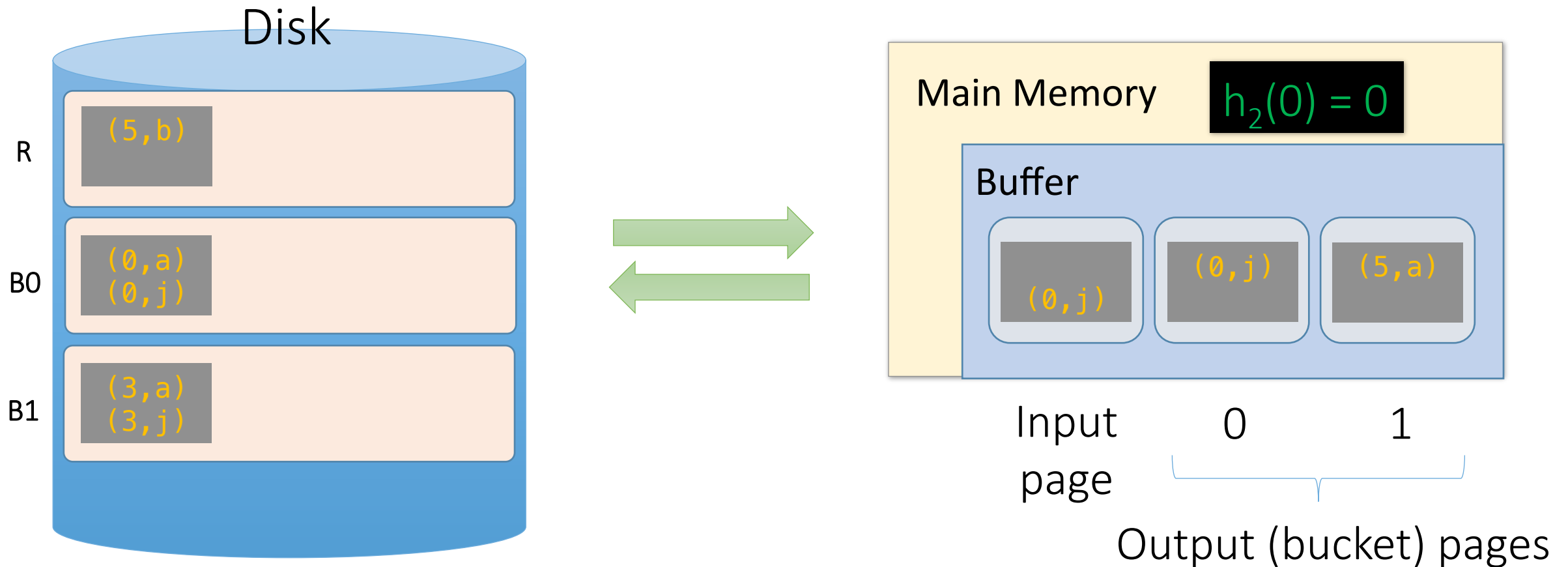
Note that collisions can occur!



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

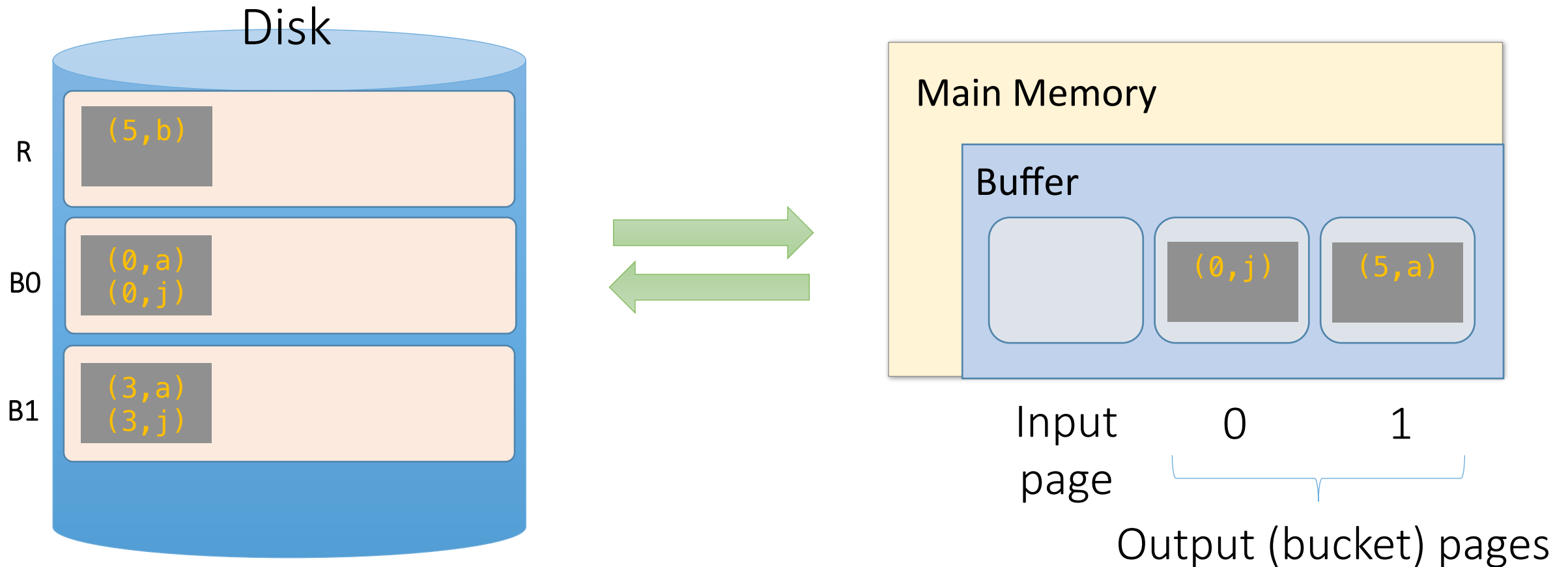
Finish this pass...



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

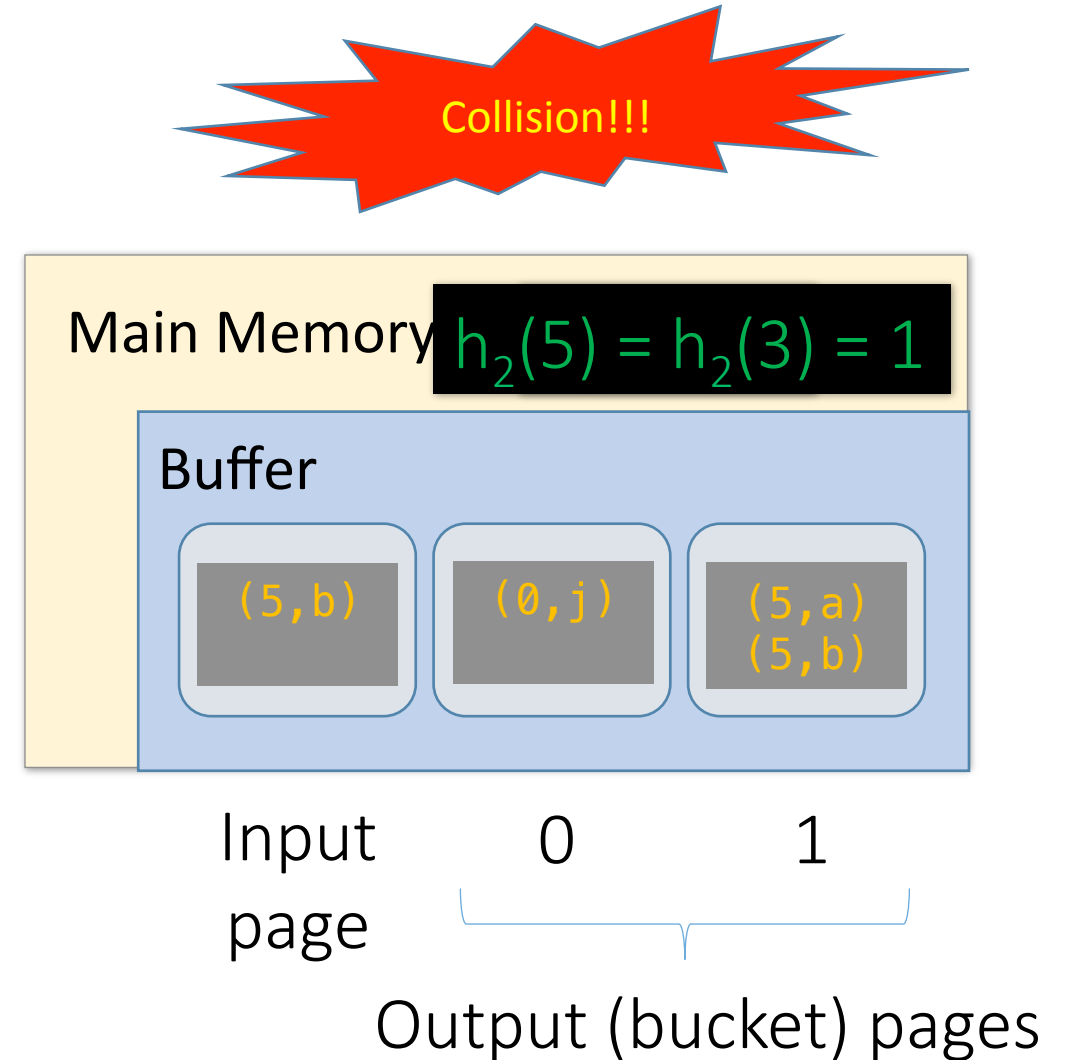
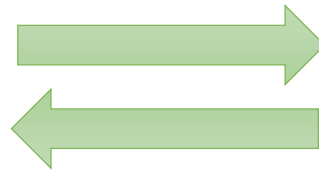
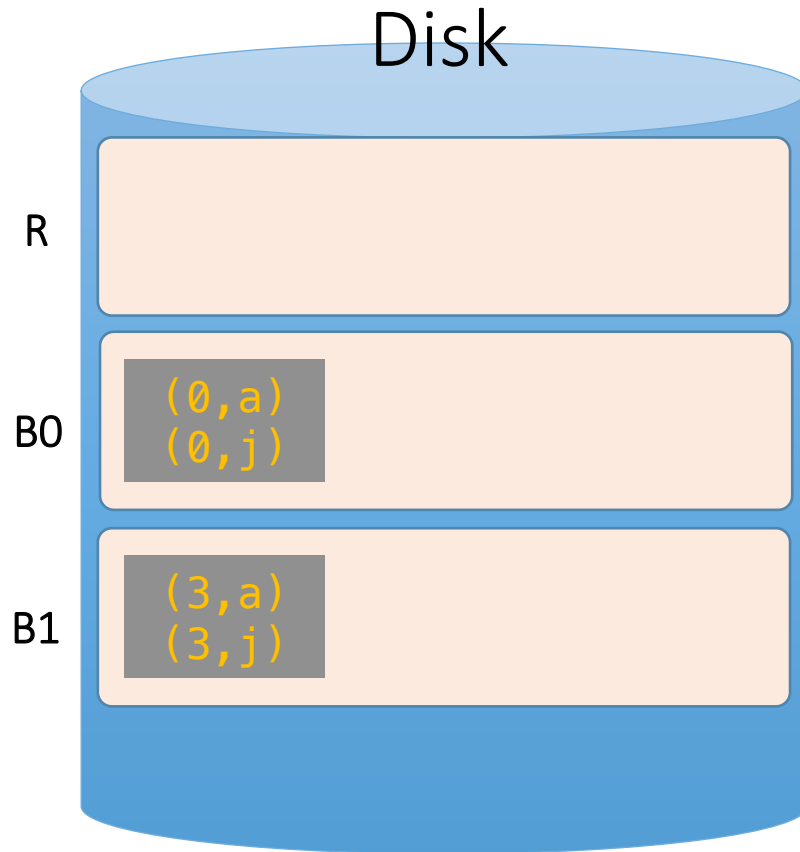
Finish this pass...



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

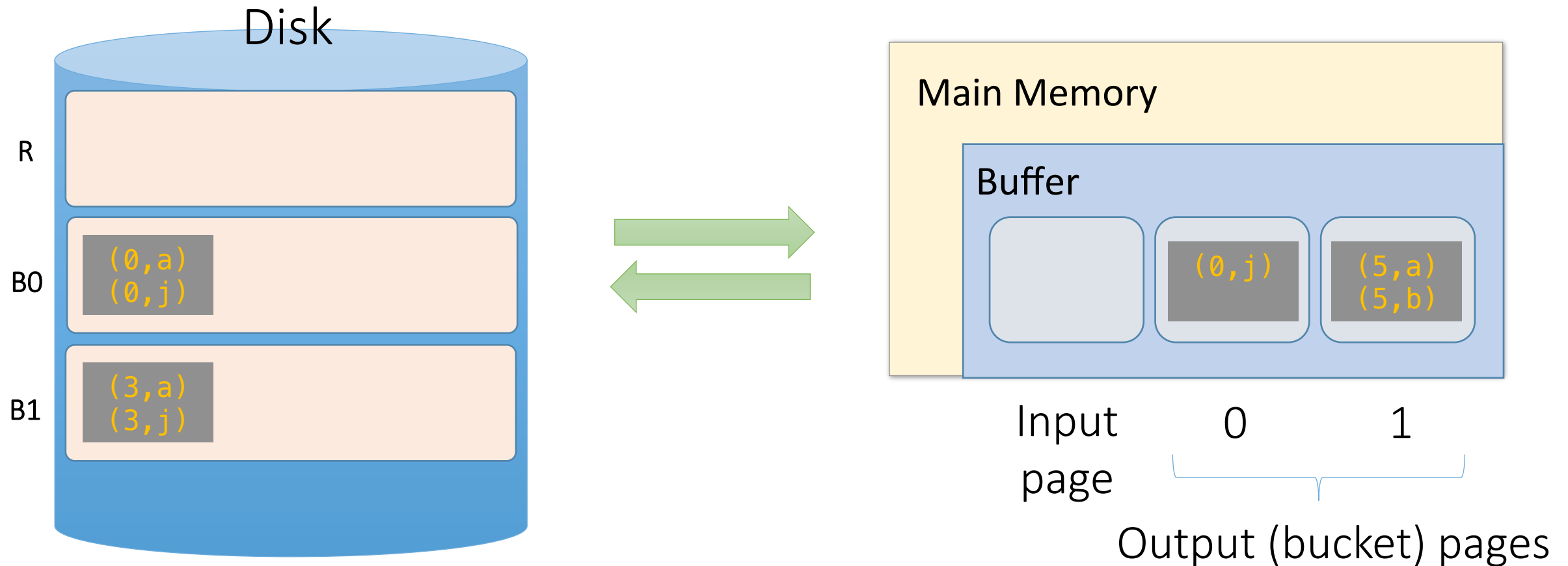
Finish this pass...



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

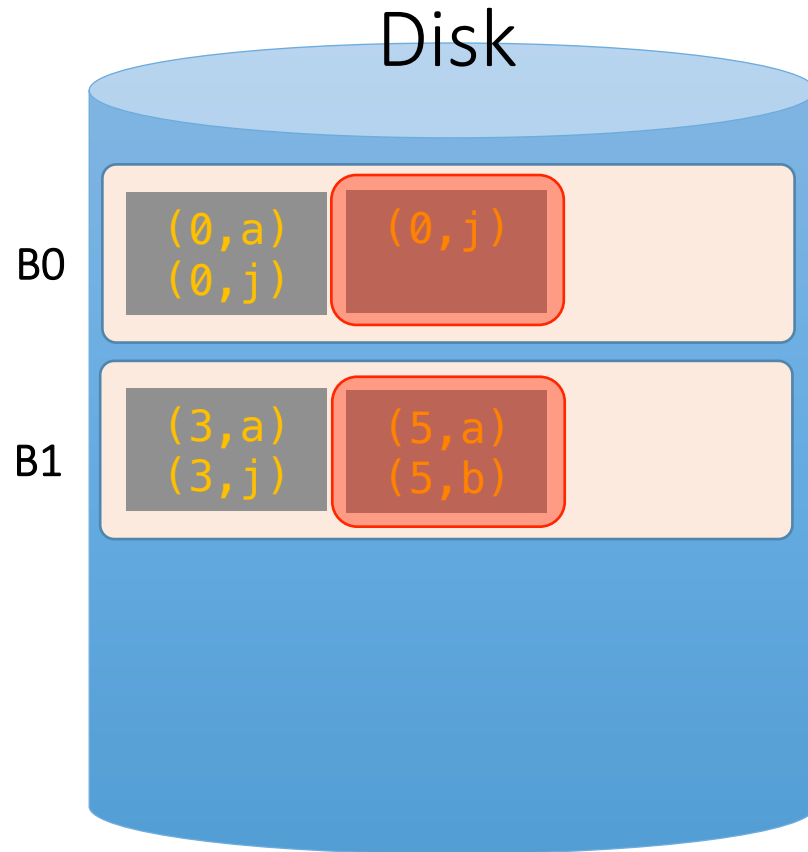
Finish this pass...



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

We wanted buckets of size $B-1 = 1...$
however we got larger ones due to:

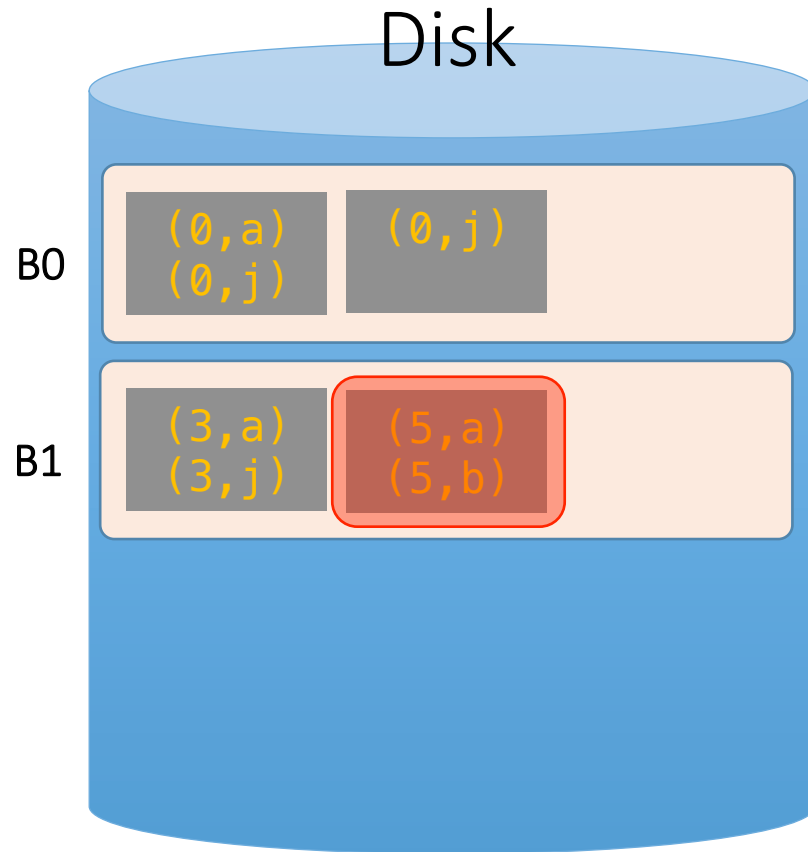


(1) Duplicate join keys

(2) Hash collisions

Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages



To take care of larger buckets caused by (2) hash collisions, we can just do another pass!

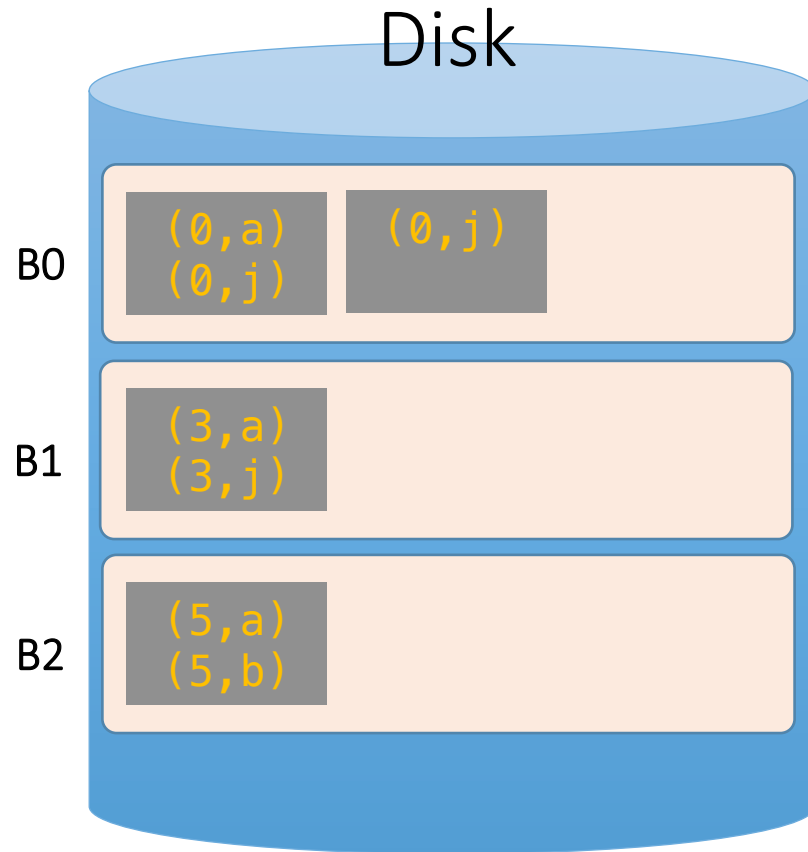
What hash function should we use?

Do another pass with a different hash function, h'_2 , ideally such that:

$$h'_2(3) \neq h'_2(5)$$

Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages



To take care of larger buckets caused by (2) hash collisions, we can just do another pass!

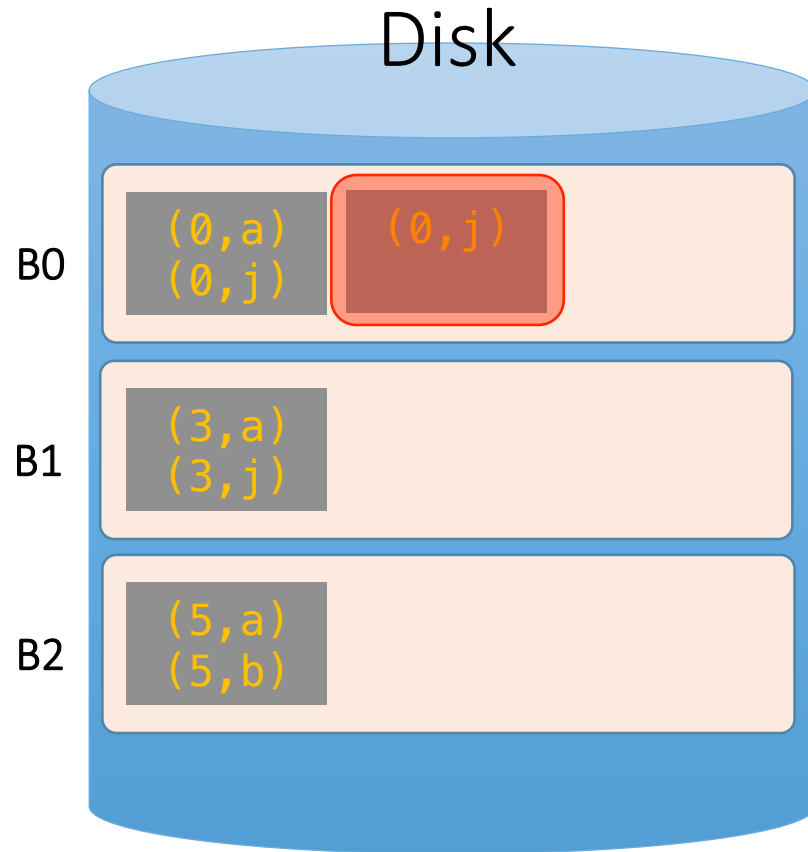
What hash function should we use?

Do another pass with a different hash function, h'_2 , ideally such that:

$$h'_2(3) \neq h'_2(5)$$

Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages



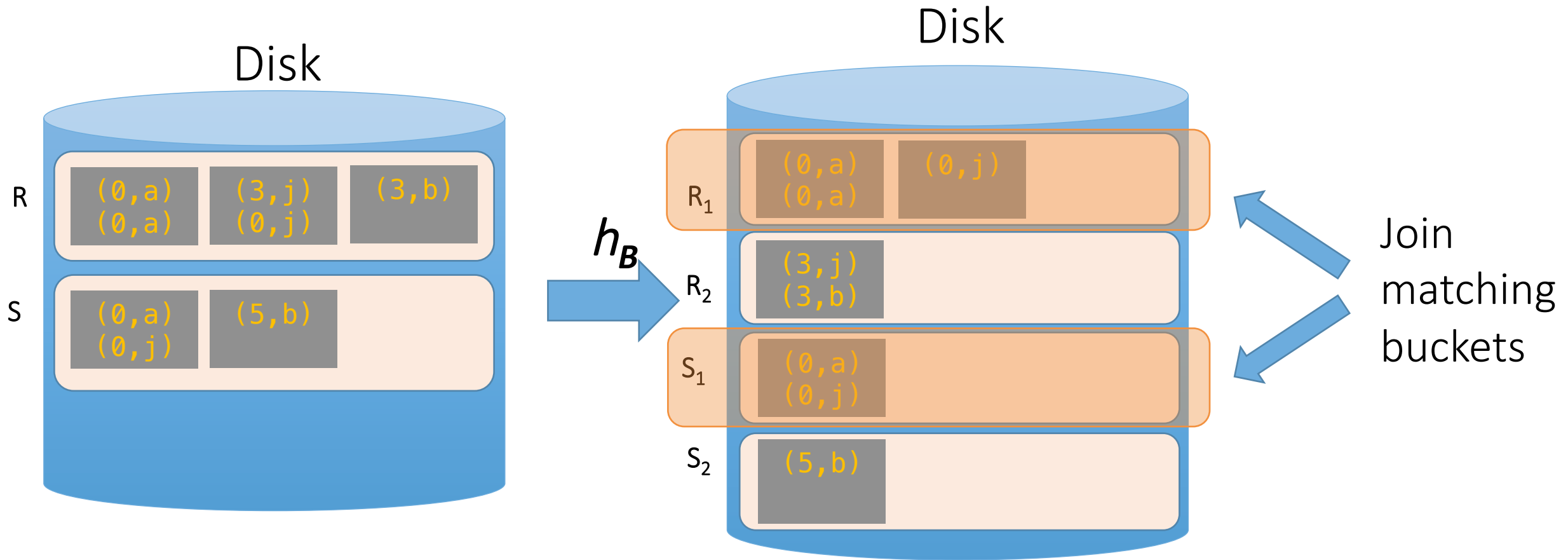
What about duplicate join keys?
Unfortunately this is a problem... but usually not a huge one.

We call this unevenness in the bucket size skew

Now that we have partitioned R and S ...

Hash Join Phase 2: Matching

- Now, we just join pairs of buckets from R and S that have the same hash value to complete the join!



Hash Join Phase 2: Matching

- Note that since $x = y \rightarrow h(x) = h(y)$, we only need to consider pairs of buckets (one from R, one from S) that have the same hash function value
- If our buckets are $\sim \mathbf{B-1}$ pages, can join each such pair using BNLJ *in linear time*; recall (with $P(R) = B-1$):

BNLJ Cost: $P(R) + P(R)P(S)/B-1 = P(R) + (B-1)P(S)/B-1 = P(R) + P(S)$

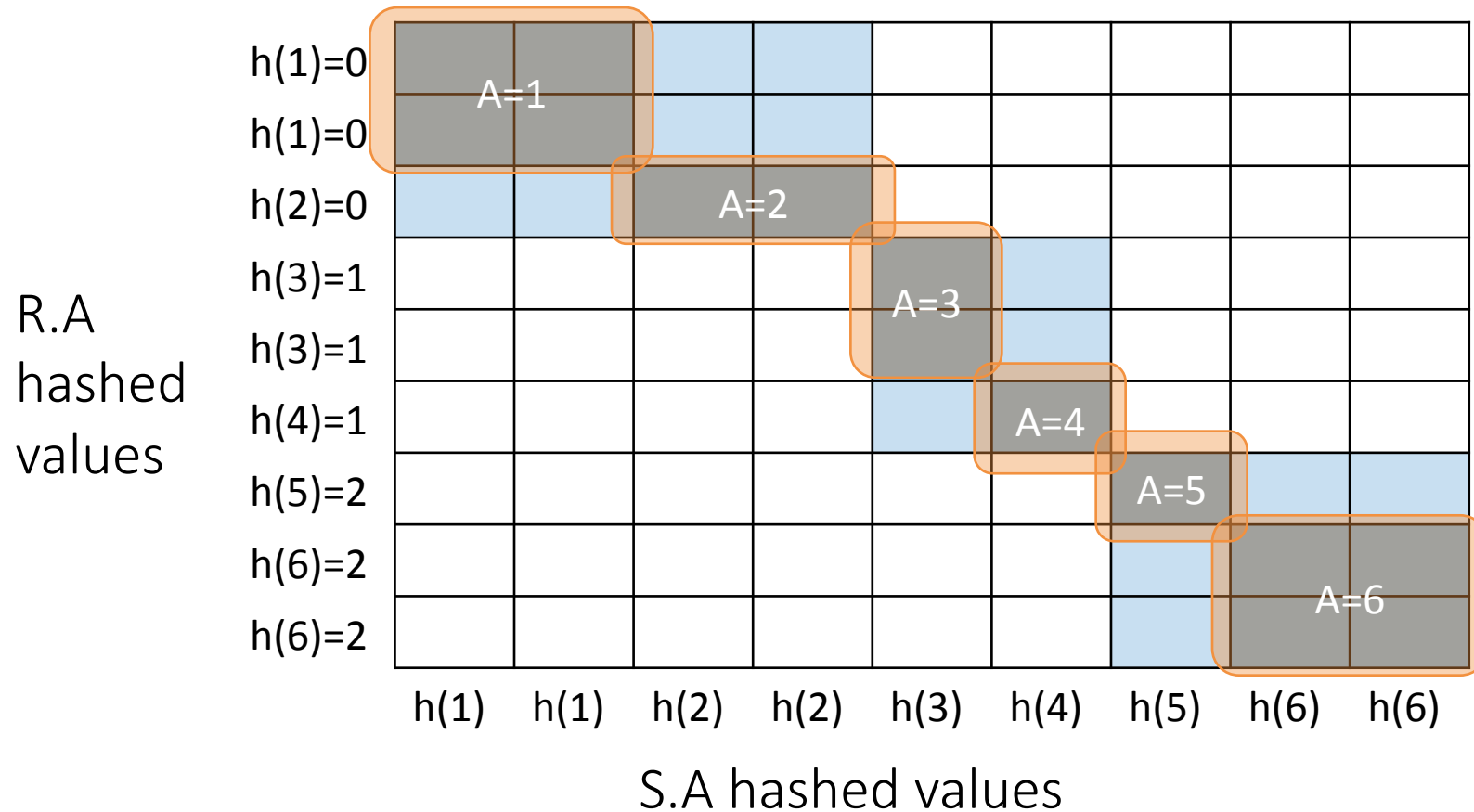
Joining the pairs of buckets is linear!
(As long as smaller bucket $\leq B-1$ pages)

Hash Join Phase 2: Matching

R.A
hashed
values

h(1)=0										$R \bowtie S \text{ on } A$
h(1)=0										
h(2)=0										
h(3)=1										
h(3)=1										
h(4)=1										
h(5)=2										
h(6)=2										
h(6)=2										
	h(1)	h(1)	h(2)	h(2)	h(3)	h(4)	h(5)	h(6)	h(6)	
	S.A hashed values									

Hash Join Phase 2: Matching

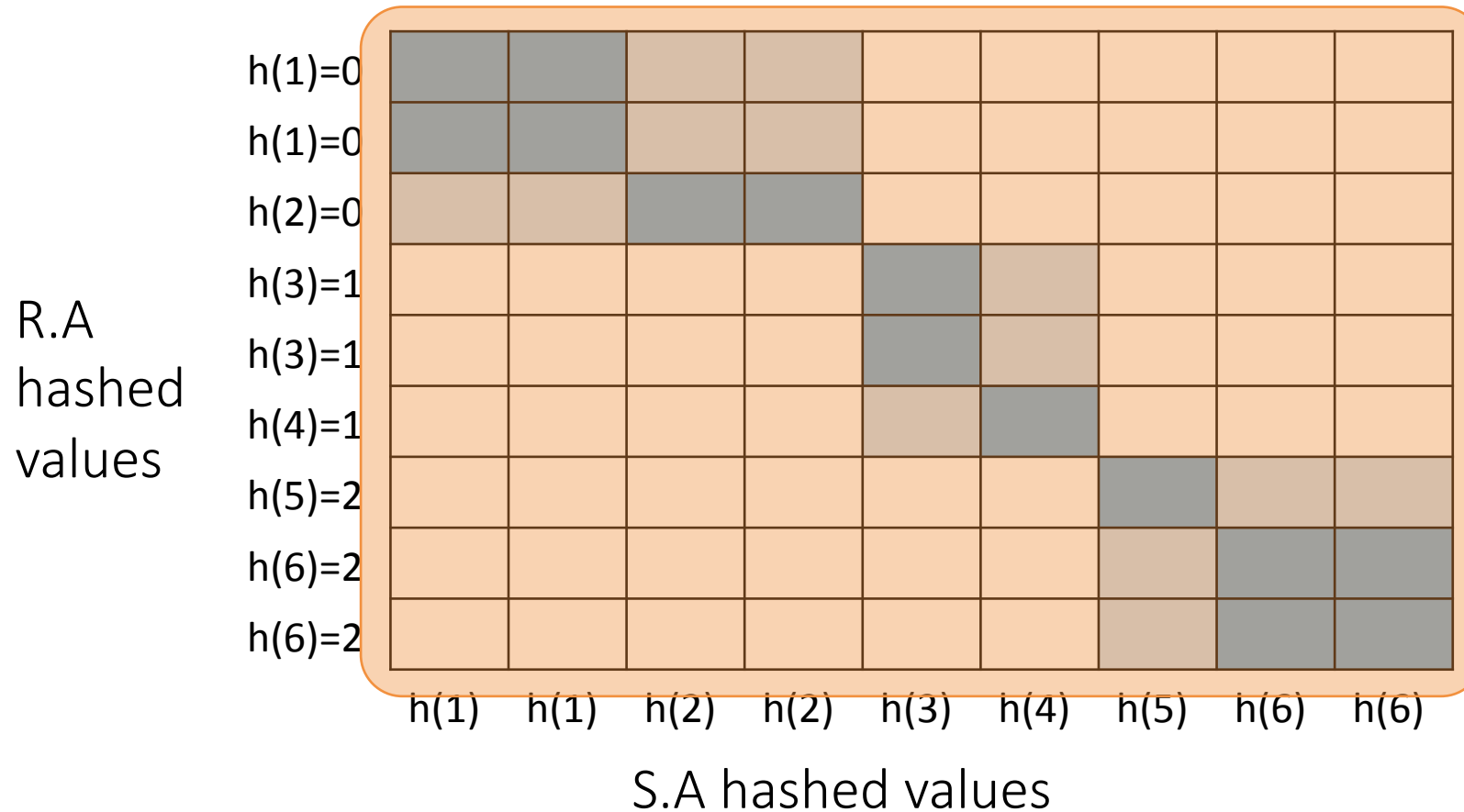


$R \bowtie S$ on A

To perform the join, we ideally just need to explore the dark blue regions

= the tuples with same values of the join key A

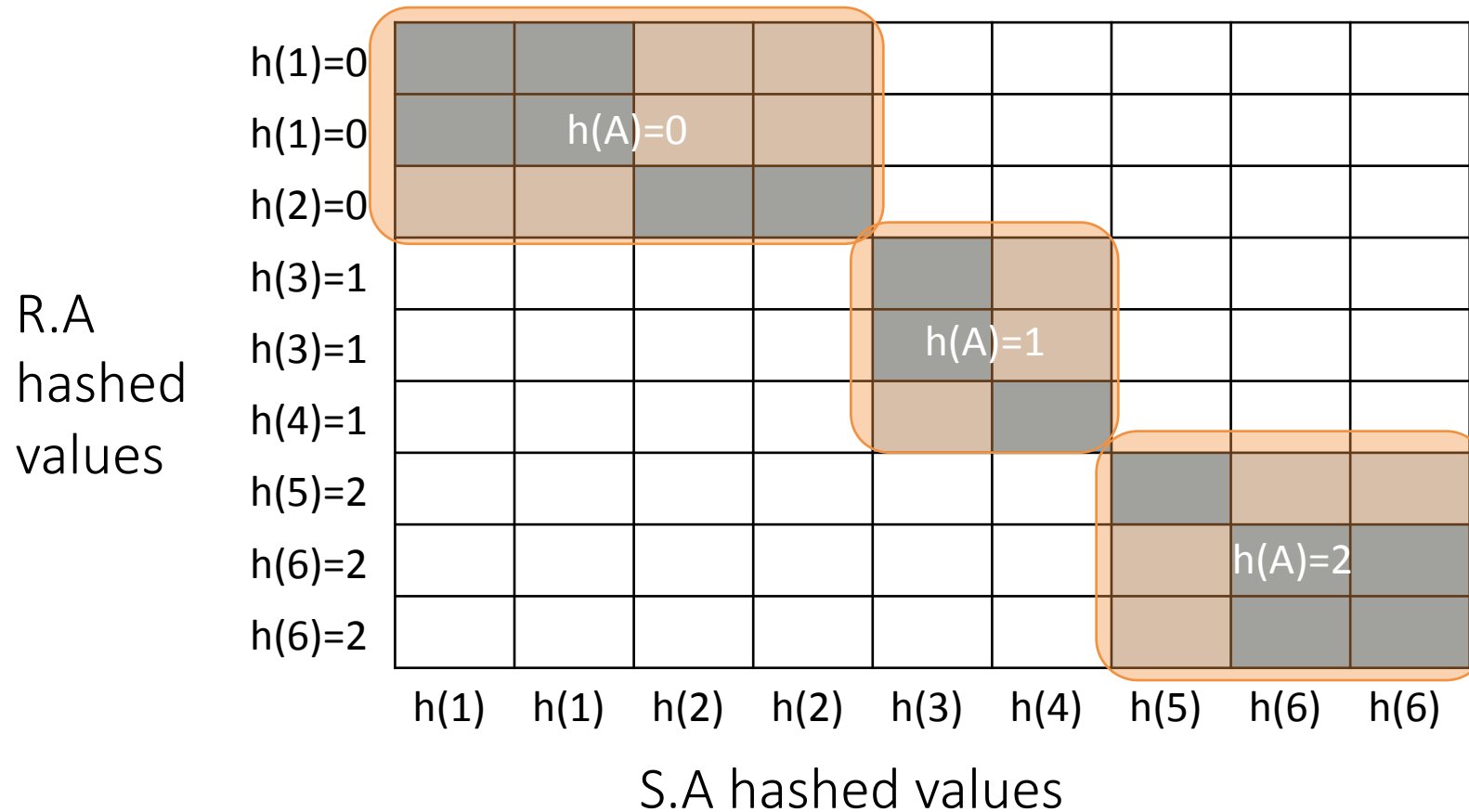
Hash Join Phase 2: Matching



$R \bowtie S$ on A

With a join algorithm like BNLJ that doesn't take advantage of equijoin structure, we'd have to explore this *whole grid*!

Hash Join Phase 2: Matching



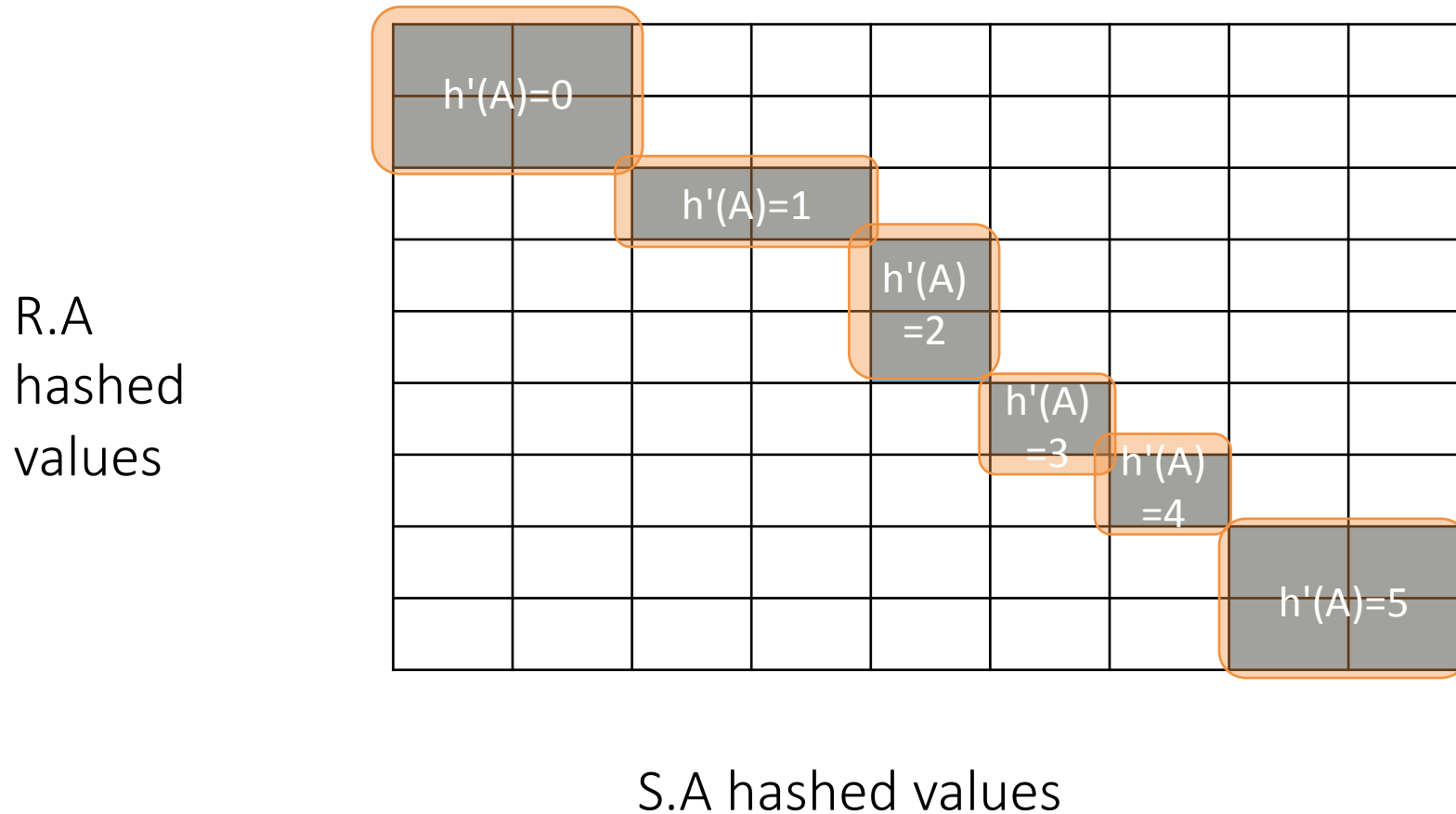
$R \bowtie S$ on A

With HJ, we only explore the *blue* regions

= the tuples with same values of $h(A)$!

We can apply BNLJ to each of these regions

Hash Join Phase 2: Matching



$R \bowtie S$ on A

An alternative to
applying BNLJ:

We could also hash
again, and keep doing
passes in memory to
reduce further!

How much memory do we need for HJ?

- Given $B+1$ buffer pages
 - + WLOG: Assume $P(R) \leq P(S)$
- Suppose (reasonably) that we can partition into B buckets in 2 passes:
 - For R , we get B buckets of size $\sim P(R)/B$
 - To join these buckets in linear time, we need these buckets to fit in $B-1$ pages, so we have:

$$B-1 \geq P(R)/B \Rightarrow \sim B^2 \geq P(R)$$

Quadratic relationship
between *smaller*
relation's size & memory!



Hash Join Summary

- *Given enough buffer pages as on previous slide...*
 - **Partitioning** requires reading + writing each page of R,S
 - $\rightarrow 2(P(R)+P(S))$ IOs
 - **Matching** (with BNLJ) requires reading each page of R,S
 - $\rightarrow P(R) + P(S)$ IOs
 - **Writing out results** could be as bad as $P(R)*P(S)$... but probably closer to $P(R)+P(S)$

HJ takes $\sim 3(P(R)+P(S)) + OUT$ IOs!

3. The Cage Match

Sort-Merge v. Hash Join



- **Given enough memory**, both SMJ and HJ have performance:

$$\sim 3(P(R) + P(S)) + OUT$$

- “Enough” memory =

- SMJ: $B^2 > \max\{P(R), P(S)\}$
- HJ: $B^2 > \min\{P(R), P(S)\}$

Hash Join superior if relation sizes *differ greatly*. Why?



Further Comparisons of Hash and Sort Joins

- Hash Joins are highly parallelizable.
- Sort-Merge less sensitive to data skew and result is sorted



Summary

- Saw IO-aware join algorithms
 - Massive difference
- Memory sizes key in hash versus sort join
 - Hash Join = Little dog (depends on smaller relation)
- Skew is also a major factor